ELSEVIER

Artificial Intelligence 86 (1996) 171-188

Artificial Intelligence

Research Note

The complexity of searching implicit graphs [★]

José L. Balcázar [*]

Departament LSI, Universitat Politècnica de Catalunya, Edifici U, Pau Gargallo 5, E-08028 Barcelona, Spain

Received July 1995; revised April 1996

Abstract

The standard complexity classes of complexity theory do not allow for direct classification of most of the problems solved by heuristic search algorithms. The reason is that, almost always, these are defined in terms of implicit graphs of state or problem reduction spaces, while the standard definitions of all complexity classes are specifically tailored to explicit inputs.

To allow for more precise comparisons with standard complexity classes, we introduce here a model for the analysis of algorithms on graphs given by vertex expansion procedures. It is based on previously studied concepts of "succinct representation" techniques, and allows us to prove PSPACE-completeness or EXPTIME-completeness of specific, natural problems on implicit graphs, such as those solved by A^*, AO^*, and other best-first search strategies.

1. The problem and the model

Heuristic search algorithms play a noticeable role among the techniques developed for attacking many relevant, practically important problems currently considered as intractable. Our aim here is to promote the use of a tool taken from structural complexity theory, namely "succinct representations", to gain further understanding of some aspects of the computational problems to which these algorithms are applied.

A common characteristic of several such algorithms, including the well-known A^* and AO^* algorithms employed in the artificial intelligence area (see e.g. [28,29]), is that the use of heuristic information, gained from study of the specific problem at hand, frequently allows for a dramatic cutdown of computer time needed to solve many a

[★] Work supported by the EC through the Esprit BRA Program (project 7141, ALCOM II) and through the HCM Program (project CHRX-CT93-0415, COLORET Network).

[*] E-mail: balqui@lsi.upc.es.

specific instance; but, on the other hand, little or no guarantee of fast performance can be given a priori (otherwise the corresponding problems would not merit anymore the adjective of "intractable").

Many combinatorial optimization algorithms and problems from operations research have been given a lot of attention from the complexity theory community, being actually a main source of examples of completeness for various complexity classes (see [9,12,14,15]). However, for optimization algorithms from AI, some of which follow analogous intuitions, much less seems to be known. Some information about problems on AND/OR graphs is readily obtained from the rich body of literature about the complexity-theoretic approach to games via alternation (let us mention in particular [31]). But for simpler search problems, all the complexity-theoretic analyses known to the author are based on concepts, tools, and conventions essentially different from those of structural complexity. This paper suggests the use of "succinct representations" to bridge this gap.

This consists of encoding the input to combinatorial algorithms by means of "programs" in some specific (most frequently, boolean) model. Encoding the input via circuits was labeled as "a natural, if not practical, representation" in [27, p. 181]. We will develop in this paper a close relative of this input encoding that is arguably practical, in that it captures the essence of the widely implemented AI approach to graph searching. Let's explain why.

Inputs to realistic graph searching

From the point of view of a naive complexity theorist, the problems solved by A* and other best-first strategies are not difficult: essentially, construction and optimization of paths in graphs, captured by nondeterministic logarithmic space complexity classes. In the AI case, the graph is a so-called "state space", characterized by the fact that the exploration of a vertex is done with knowledge of additional information, gathered along the path to it or computed on the spot. The algorithm may decide to expand a vertex, and this operation consists of computing all its successors. In most cases, what is wanted is to find a path from a start vertex to one of a set of goal vertices, often with the additional request of minimizing a cost function.

Actually, starting from Dijkstra's algorithm, and continuing with many other applications of algorithm design techniques, polynomial time algorithms solving that sort of problems exist, offering good practical performance. Most of them are currently available, precompiled in libraries, ready to use very easily, as executable subroutines of C++ programs [22]. Variants of A* can be seen as refinements of Dijkstra's algorithm through the use of so-called "admissible" heuristic information (see the discussion in [20]).

But from the complexity-theoretic approach, one would consider that the input to the algorithm is the whole graph, and this would be already unacceptable to practitioners: in most practical cases, the search graph is huge enough that even linear time on its size is infeasible by far. This accounts for the labeling of "hard", often applied to the problems solved by such AI algorithms, and for the fact that these algorithms can be employed to attack well-known NP-complete problems. It is a task for complexity theory (which we address here) to find a framework to analyze these problems in a more satisfactory way, taking into account the astonishing sizes of the searched graphs.

There are several results on the formal analysis of the complexity of these algorithms; see [28] (in particular, Chapters 5 and 6). The context is often probabilistic, the analysis is given mostly as a function of the length of the optimal output path, and made in terms of other quantities, such as the number of different expanded vertices, or the total number of vertex expansions (which in A* is exponentially higher since, in the bad context of a nonmonotonic admissible heuristic, the same vertex may be chosen for expansion exponentially many times). Such frameworks do not allow for easy translation into standard complexity classes. In other cases the analysis is restricted to the use of the algorithm to solve some specific, particular problem.

We want to supplement the view obtained from the probabilistic average analysis, which is of course very useful but depends on several simplifying assumptions and on the hypothesis of some specific probability distribution [28]; we will see how to model these algorithms in such a way that the standard complexity classes give informative properties of (the problems solved by) these algorithms, in as much generality as possible, and in terms of worst-case complexity.

Our point of view is that, in order to obtain a consistent approach, informative for workers of other areas, one must consider that the input to such search algorithms is *not the state space graph*, but is instead the (hopefully efficient) *procedure for vertex expansion*. Note that, if simply to expand a vertex is computationally expensive, then the mere feasibility of all such algorithms is at stake. Moreover, this condition that expansion better be efficient implies that the outdegree of the state space graph must be appropriately bounded, lest simply the production of a too long list of successors be unaffordable.

It is not clear a priori how complexity classes could cope with the idea of having a vertex expansion procedure as input. As it turns out, complexity theory has the right intuitive tool to put to use: here we suggest to resort to an appropriately refined notion of succinct representation.

Indeed, one could argue that the vertex expansion procedure is, in a sense, a description of the whole graph, allowing for efficient local treatment. One well-studied form of succinct representation is based on encoding the transition matrix: a small, fast enough procedure receives as input (the codes of) two vertices and outputs simply the bit indicating presence or absence of an edge between them (or even, more elaboratedly, its cost). But this is not appropriate here: the search algorithms we want to model frequently assume that more information can be obtained from each vertex, including the complete list of successors. It may be infeasible to extract this list from the transition matrix, for instance when the number of actual successors is not high but the total number of vertices is large.

We propose here to use as succinct representation of the graph the formalization of a procedure that, given a vertex, produces the list of successors, including, when appropriate, the cost corresponding to each edge. It must be feasible, so certainly a mild condition on it is to require it to work in polynomial time, which marks a weak feasibility limit. Therefore, we assume that it can be implemented with a family of boolean circuits: it is well known that this model exactly characterizes (via completeness) polynomial time computations [1]. Let us set this idea in the perspective of previous research on succinct input representation.

Problems on succinct representations

The study of algorithmic graph problems on succinct representations was introduced independently in [8,32], and has been proven useful in other applications of complexity theory. Essentially, the idea is as follows: while traditional models of computation assume their input represented, under some reasonable encoding, as a string of letters over some alphabet (frequently binary), it is interesting as well to see what happens to computational problems when their instances are encoded in some other, hopefully more compact, form.

A paradigmatic case is that of graphs with regularities, which arise in many practical and theoretical areas of computer science: representing highly regular graphs by standard means such as transition matrices may result in an undesirable waste of memory space, since these regularities may allow for much more concise representations in the computer memory. Schemes which allow considerable savings include hierarchical representations and boolean computational models [8,21,32]. The same idea can be applied to other data types via integer expressions [32] or vector languages [19], and an essentially analogous intuition (with a very different technical development) was used in [13] to obtain complete problems for deterministic linear space. We will use here a variant of the boolean circuit representation.

In principle, the intuitive consequences of such input conventions are contradictory: being, in general, shorter than the full description, they allow for less running time (which is a function of the length of the shorter input); but since only very regular instances allow for substantial savings in the encoding, it may be the case that the algorithms operate faster on these instances. So, a careful and detailed analysis of each input convention is necessary.

In the case of decisional problems on graphs, the input is not the graph itself but (the encoding of) a boolean circuit which computes its transition matrix. For this model, [8,32] classify as complete in some class of the polynomial time hierarchy or the counting hierarchy (or higher up) many specific problems on succinct instances; many of these are polynomially solvable for standard input representation. Then [27] pointed out that the use of projection reducibility allows one to prove that all known NP-complete problems become NEXP-complete under succinct input representation via circuits (see also [24, Ch. 20]). This observation is extended in [2], by means of log time reducibility, to a very general result (the Conversion Lemma) relating the complexity of problems on standard input convention to their respective succinct versions, independently of the complexity classes in which they lie. This last result also abstracts from the previously used graph-theoretic setting, since the Conversion Lemma works on binary encodings of arbitrary data types.

Subsequently, the notion of succinct representation and variants of the results from [2] have been employed in [4,7] to characterize the computational complexity of a number of problems from logic programming and databases; specifically, the combined complexity of queries based on default logic and certain issues arising from disjunctive datalog queries under various semantics, respectively. Similar technical tools and intuitions, based on comparing the succinctness of the descriptions under different models of representation, appear also in [3,10], where they are used to obtain results in other settings.

This tool could be used as such for our study of graph search algorithms, but would admit a serious criticism: the model assumes a restriction that, in general, does not apply to practical cases. Indeed, in the work done so far one assumes that the input circuit is only able to indicate the presence or absence of an edge, given both endpoints; practical applications assume the ability to expand a vertex, i.e., generate all its successors, and this may be infeasible from the plain "transition matrix" implementation. A better alternative is to introduce a new model for succinct inputs, better suited to the application we have in mind, which closely parallels the idea of giving, as input to the algorithm, a procedure to expand nodes. We will see that this approach gives more informative characterizations of the hardness of graph search problems.

2. Preliminaries

Most of our notions of complexity theory regarding models of computation, complexity classes, and reducibilities, are standard [1,24]. We assume decisional problems to be encoded as sets of strings over the standard binary alphabet. Likewise, functional problems are encoded as partial functions from strings to strings. It is a well-known fact (heavily employed when computers are in use) that binary strings can represent numbers and many other combinatorial structures such as graphs. For a string x, we denote by $|x|$ its length. Real-valued functions are assumed to be rounded to the nearest positive integer. Graphs are assumed to be directed. We assume that $\langle .,. \rangle$ is an easily computable pairing function, and that both arguments can be easily computed from the result.

Our model of computation is a variant of the multitape Turing machine; it is well known that other more realistic models are equivalent to this one modulo polynomial time overheads and constant factor space overheads, so that all the complexity classes we mention are invariant with respect to the machine model employed.

The slight difference between the standard Turing machine model and ours consists in that, since we will need to work with sublinear time computations, we cannot afford sequential access to the input. Instead, we use indexing machines [5], in which there is a specific "indexing tape" which points to the input tape. On an input of length n, when the machine enters a specific "read" state with $i \leqslant n$ written on the indexing tape, then in one step the ith input symbol is transferred to the head of the first worktape. A special "error" mark appears there if the contents of the indexing tape is $i > n$. Note that this is nothing but a straightforward formalization of the idea of direct access to the input, as if it is stored in RAM.

We will mention the following generic complexity classes of decisional problems: for space or time bounds f, respectively, we have DSPACE(f), corresponding to problems A such that the problem of deciding whether $x \in A$ that can be solved within memory space $f(|x|)$, and DTIME(f), corresponding to problems A such that deciding whether

$x \in A$ that can be solved within computation time $f(|x|)$. Many classes have more familiar names, and thus

$$\text{LOGSPACE} = \text{DSPACE}(\log n), \qquad \text{P} = \bigcup_{k\in\mathbb{N}} \text{DTIME}(n^k),$$

$$\text{PSPACE} = \bigcup_{k\in\mathbb{N}} \text{DSPACE}(n^k), \qquad \text{EXPTIME} = \bigcup_{k\in\mathbb{N}} \text{DTIME}(2^{n^k}).$$

We will briefly mention in the wrap-up discussion the well-known class NP, which is defined by polynomial time nondeterministic machines.

We also need two concepts of reducibility. The first one is standard: a decisional problem A is polynomial time m-reducible to a problem B, denoted $A \leqslant_m B$, if there is a polynomial time computable function f for which $x \in A \iff f(x) \in B$ holds for all x. So f provides a way of solving A in polynomial time given any arbitrary way of solving B in polynomial time.

Our main results are based on polylog time m-reducibility. A function f is computable in polylogarithmic time if there is an indexing machine that, on input x and j, with $j \leqslant |f(x)|$, outputs the jth bit of $f(x)$ in time $\log^{O(1)}|x|$. Analogously to the polynomial time case, a polylog time reduction from problem A to problem B is a function f computable in polylogarithmic time such that, for all x, $x \in A \iff f(x) \in B$. We say that A is PL-reducible to B, and denote it $A \leqslant_m^{\text{PL}} B$. The use of polylog time instead of log time reductions in combination with succinct representations was one of the contributions of [7].

The concept of reducibility gives rise immediately to the concepts of hardness and completeness for a complexity class. A problem A is hard for a class $\mathcal{D}$ under a reducibility if all problems in $\mathcal{D}$ are reducible to A. A problem A is complete for a class $\mathcal{D}$ if it is hard for it and also belongs to it.

From the time and space hierarchy theorems of complexity theory, it is well known that LOGSPACE $\neq$ PSPACE and that P $\neq$ EXPTIME. In particular, this implies that a complete problem for PSPACE, under any of our reducibilities, cannot be in LOGSPACE, and actually cannot be solved in subpolynomial memory space (here subpolynomial means $o(n^\alpha)$ for all α). Similarly, a problem that is complete for EXPTIME cannot be solved in polynomial, nor even subexponential, running time. As of now, it is possible that P = PSPACE, so that completeness for PSPACE does not rule out the possible existence of a polynomial time algorithm; but such an algorithm would imply immediately polynomial time algorithms for all problems in PSPACE, including all NP-complete problems and many others seemingly harder than them.

Let f be a polylog time reduction. We say that f is polylog time covered (or PL-covered) if the values j such that the bit $\langle i,j\rangle$ of $f(x)$ is 1 are predictable in advance, only knowing i and $|x|$, as well as the size of $f(x)$; more formally, if there is an indexing machine that on inputs i and n (in binary) and in polynomial time (equivalently, polylog time when inputs are in unary), computes a value m and a list L such that, whenever bit $\langle i,j\rangle$ of $f(x)$ is 1 for some x of length n, we have $j \in L$ and $m = |f(x)|$. Of course, this definition only applies to reductions for which the length of the output only depends on the length of the input, and not on its actual value.

This notion will be used in the following context: the output of f will be interpreted as an instance of a decisional problem on (directed) graphs, described by a transition matrix. Bit $\langle i,j\rangle$ indicates presence or absence of the edge $\langle i,j\rangle$ in the output graph. Coverability means, therefore, that, when all x with $|x| = n$ are considered, and for each i, there are only $\log^{O(1)} n$ many potential outgoing edges from vertex i that may appear in any of the graphs $f(x)$, and that all their other endpoints are easily computable. This technical condition is necessary in some of our statements, and can be extended easily, if necessary, to the case in which the graphs are weighted.

3. A new model of succinct instance representation

Following the informal description given in the introduction, we now choose to represent graphs as follows. First, we restrict ourselves to graphs without isolated vertices. For a graph of less than n vertices and degree d, its representation is a boolean circuit of $\log n$ inputs and $d \log n$ outputs. Without loss of generality, we assume that the value of n is encoded or hardwired into the circuit, so that it can be read out easily. This can be achieved with $\log n$ clearly identified constant gates, and would correspond, intuitively, with a constant declared inside the vertex expansion procedure. Vertices are numbered, reserving code 0 to be an additional dummy vertex name. Assume that vertex v_k corresponding to number k has as successors $v_{i_1} \dots v_{i_p}$ with $p \leqslant d$; then p among the d groups of $\log n$ output gates must each give one of the numbers i_j, written in binary with zeroes to the left up to size $\log n$, and without repetition; the remaining outputs must be all zero, i.e., describe the additional dummy vertex.

Clearly the representation is not unique, but given a circuit c of $\log n$ inputs and $d \log n$ outputs, there is a single graph G_c represented by it: take as vertices the positive integers that can be written with $\log n$ inputs, set edges by feeding each to the circuit to find out the successors of each vertex and discarding vertex zero whenever it shows up, and finally remove any isolated vertices that might be left. When c is a syntactically incorrect circuit in which the number of outputs is not a multiple of the number of inputs, and which therefore does not represent any graph, then it is mapped to some fixed, trivial graph $G_c = G_0$.

Note the following fact: it is not necessary that the vertices are numbered by consecutive numbers. As a consequence of our convention, we may afford as many numbers as convenient that do not correspond to any vertex. It suffices that the circuit is able to detect them, to output an empty list of successors, and to never output them as successors of other vertex numbers. In this way, they are represented as isolated and will be discarded by our interpretation.

Let A be any decisional problem on graphs. We denote by sA the succinct version of A, defined as follows: $sA = \{c \mid G_c \in A\}$. We state now the main property of this representation, from the complexity-theoretic point of view. The analogous result for another, technically different, simpler form of succinct representation was given in [2] (knowledge of that proof may help in understanding this one); it used also a slightly different reducibility, and did not need the extra condition of coverability. The proof is now more complex due to the more involved definition of succinct representation, which now must provide more information.

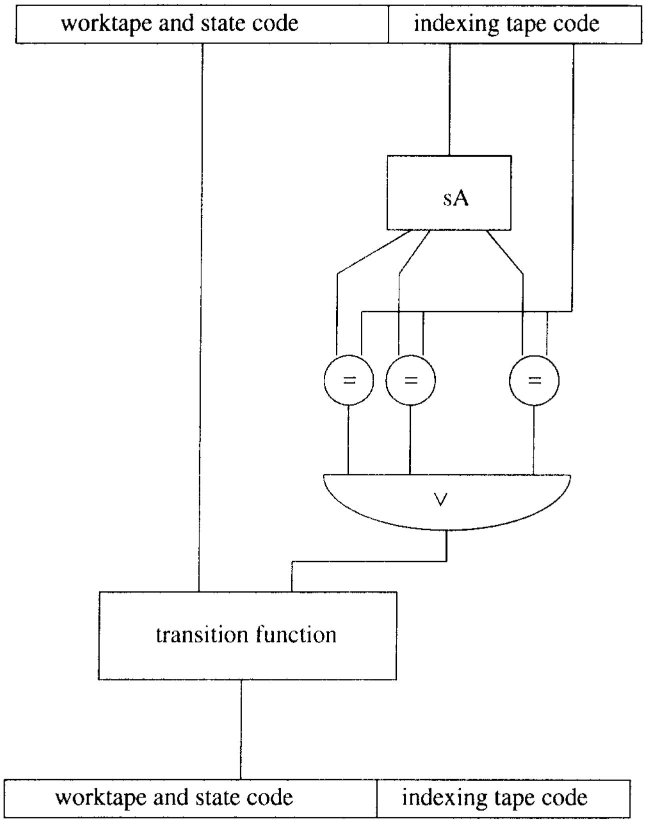

Fig. 1. Circuitry between two layers in the simulation of M by C'.

Theorem 1. *Let A and B be decisional problems defined on graphs. If $A \leqslant_m^{\mathrm{PL}} B$, via a PL-covered reduction, then $sA \leqslant_m sB$.*

Proof. Let f be the reduction, and let M be the indexing Turing machine that computes it. We must design a reduction between the corresponding succinctly represented problems. Assume M works in time $\log^k n$. By the definition of computing in polylog time, given an instance of A, M finds individual bits of the image under the reduction. Therefore, its input is an instance of A, i.e., (an encoding of) a graph G on, say, n vertices, and a pair of vertices of $f(G)$; it finds whether this pair is indeed an edge of $f(G)$. Via a now standard transformation (see [1]), and assuming that all the bits of G are available at all times, a boolean circuit can be constructed which computes any requested bit of the output of the reduction on x. The size of the circuit is $\log^{2k} n$. However, the bits of G are not so readily available.

What is available now, instead, is an instance of sA: a circuit again, which, on input j, provides the list of all vertices that are successors of vertex j in G. Without loss of generality, we have assumed that n can be easily read out from this input circuit.

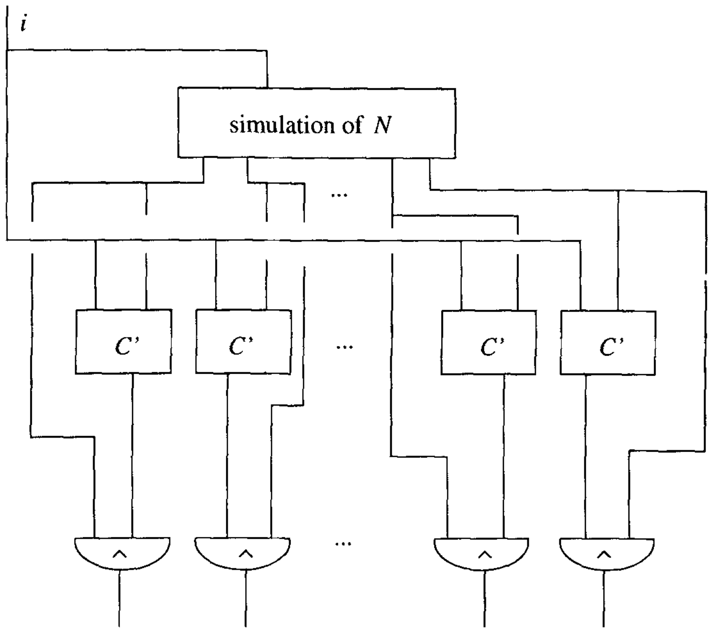

Fig. 2. Overall structure of the constructed succinct representation.

Examination of the circuit simulating M (see [1]) shows that it consists of layers, each layer t computing the configuration of the machine after t steps. Each layer contains gates whose output bits encode the contents of the indexing tape, and therefore mark the bit being read at that step. The next layer needs knowledge of that bit. Therefore, between each pair of layers, extra circuitry is added to obtain that bit, as follows: if the indexing tape requires the bit corresponding to edge (p,q) of A, then p is fed into the input circuit (the given instance of sA) and q is matched for equality (in parallel) against all the output vertices of the input circuit. This provides exactly the needed bit. Fig. 1 describes intuitively the circuitry between consecutive layers; the top and bottom boxes represent a large set of parallel wires carrying the sequence of bits encoding the configuration, where we have assumed that the encoding of the indexing tape comes at the end.

In this way, the whole circuit calculating each bit of $f(G)$ is obtained; call this circuit C'. It would qualify as a succinct representation of $f(G)$ under the definition of [8]; but does not yet, for ours, exactly for the same reason that the previously used succinct representations did not provide an adequate model for the graph search algorithms: it computes a bit of the transition matrix, not a list of successors.

Here is where coverability enters into play. Let N be the machine that computes the covering. Note that it provides also the size of the output, that we need to hardwire into the circuit for that output. We construct the circuit we finally desire, C, as follows. Let i be the input to C. Using the same simulation as before, now applied to the machine N, and run simultaneously with the simulation above, in parallel, on n and i, a subcircuit can now produce a superset of the list of successors of vertex i. Each of them, k, together with i itself, is fed into its own replica of C', and if the output is 1 then k is transmitted through to the output of C. See Fig. 2.

In this way, C computes exactly the list of successors of i in $f(G)$, i.e., it is now a bona fide succinct representation of $f(G)$, constructed on the basis of a succinct representation of G. This is exactly what the reduction from sA into sB must construct. All the computations involved are easily seen to be accomplished in polynomial time; this completes the proof. □

A straightforward modification takes care of the case in which weights (or costs) are associated to the edges. Our only proviso is that the costs are not exceedingly high; more precisely, since our application domain requires to consider already the size of the graph N too high, we require that the weights can be written down in polylog many bits (on N). In this way, the polylog time reduction is able to provide not only the presence or absence of an edge but also the cost associated with it. Then, both A and B can be defined in terms of the costs, and the relationship will still hold.

If the succinct representation of a graph becomes exponentially smaller, then the complexity of a problem A may jump up by about one exponential when passing to sA. It is not difficult to see that it does not jump up more:

Lemma 2. *If* $A \in \mathrm{DTIME}(f(n))$ *then* $sA \in \mathrm{DTIME}(f(2^{O(n)}))$, *for* f *at least linear; if* $A \in \mathrm{DSPACE}(f(n))$, *then* $sA \in \mathrm{DSPACE}(f(2^{O(n)}))$, *for* f *at least logarithmic; and similarly for nondeterministic complexity classes.*

The proof is a simple modification of the analogous one in [8]: for time classes, given the circuit, first expand it into the whole graph, then apply an algorithm on the standard representation; for space classes, input bits are computed from the circuit (and possibly recomputed many times) whenever necessary, then discarded.

We end this section with the following technical fact, necessary in later proofs: the "succinct representation" operator has a kind of "inverse", given by what we will call *trivial graph* representation. Let w be a binary string. Then $\mathrm{tg}(w)$ is a "chain" graph of m nodes, each but the last having a single edge to the next, the last to itself, and m being selected so that its binary representation is $1w$. Note that it is exponentially large in $|w|$. Overloading the operator, let $\mathrm{tg}(L) = \{\mathrm{tg}(w) \mid w \in L\}$.

Lemma 3. *Every set* L *of binary strings is m-reducible in polynomial time to the succinct representation of* $\mathrm{tg}(L)$.

Proof. It suffices to construct from w a circuit corresponding to $\mathrm{tg}(w)$. It simply checks whether the number of the input vertex is exactly $1w$ (read in binary), which corresponds to the last vertex, to output the same number; or it is less than $1w$, to add one and output the result. Else, it outputs 0, i.e., marks it as an isolated vertex, which is ignored by the convention on succinct representations. □

4. Application to graph search

The graph accessibility problem (GAP) is: given a directed graph, a source vertex, and a goal vertex (or set of them), decide whether there is a path connecting the source vertex to a goal. In the weighted variant, costs are associated to the edges; an integer k is also given, and the question is whether such a path of total cost at most k exists. Without loss of generality, we assume that the source and goal vertices have always specific numbers (such as 1 and 2 respectively), so that the input to the problem is just the graph.

Some broadly employed heuristic search algorithms solve precisely this problem, with the peculiarity that the graph is *not* given explicitly. Instead, a procedure is given to "expand" a node, i.e., to find the list of all successors of a given vertex, together with the costs of the edges in the weighted variant.

Vertices are described, in these cases, by some kind of "configuration" or specification of "how the vertex looks like" (e.g. pieces on a board); essentially, the information that must be present in this description is (at least) the necessary data to compute feasibly all successors. We assume that the number of vertices (or configurations) may reach an exponential on the number of bits used to store a configuration.

The point is now that the internal representation of the configuration in a computer program, whatever means are used for it, can be read as a binary number (a form of Gödel numbering, in a sense) and directly interpreted as *the* number of the vertex. Thus, the vertex number abstracts the description of "how the vertex looks like" and contains information about how to find the successors. Syntactically incorrect descriptions correspond to numbers that do not represent vertices, and will be treated as isolated vertices by the circuit representation, as indicated in the previous section.

We assume that the "expand" procedure is feasible, i.e., takes time polynomial on the size of (the codenumber of) the vertex. Every reasonable application of graph search will need this condition; otherwise the search will be infeasible just because some node expansions, to start with, take far too long. This also implies that the degree, and the number of bits needed to specify costs, are also polynomial on the size of the number of the vertex. Note that this upper bound may be as restrictive as a polylog function on the size of the whole graph; in concrete applications, constant degrees and cost sizes are frequent.

Equivalently, under this condition, we can formalize the expansion procedure as a boolean circuit, via the standard simulation of polynomial time computations by boolean circuits. This simulation is feasible, and any other sequential programming language can be assumed instead for the expansion procedure, since all these can be compiled into circuits in polynomial time. This circuit is the input to a search algorithm based on node expansions. The output wanted is given, in general, by an optimality condition, but a (somewhat artificial) decisional problem can be associated to it in a standard way [9]; and the optimization problem is no easier (and possibly harder) than the decisional problem.

In this case, the decisional problem is exactly (the weighted version of) sGAP, since the solution is a path and the input is a circuit describing the vertex expansion procedure to construct the graph. Our aim in this section is to use the setup from the previous section to show:

Theorem 4. *sGAP is* PSPACE-*complete.*

The following lemma is needed for the proof:

Lemma 5. *GAP is hard for* LOGSPACE *under PL-covered polylog time m-reductions.*

Actually, GAP is complete in the larger class corresponding to nondeterministic logarithmic space under logarithmic space reducibility [16,30]. The proof of the lemma is obtained by an analysis of the completeness proofs in these references, checking that the necessary computations can be performed in polylog time on indexing machines; we simply sketch it here. Essentially, each node is a configuration of a Turing machine M, and an edge connects successive configurations: this amounts to checking locally constant size fragment of each configuration to test the agreement with the transition function. The edges depend on the symbol read from the input at that configuration, but it is possible to compute all possible successors over the finitely many possibilities for that symbol: this proves coverability.

Now let's go back to the proof that sGAP is PSPACE-complete.

Proof. Membership in PSPACE follows by an argument like that of Lemma 2. Let us argue hardness. Let L be an arbitrary set in DSPACE(n^k). Let $L' = \{w10^{|w|^k} \mid w \in L\}$, so that $L \leqslant_m L'$. Also, by Lemma 3 above, $L' \leqslant_m s(\mathrm{tg}(L'))$. It is immediate to see that $\mathrm{tg}(L')$ is in LOGSPACE, just by rejecting all graphs that are not $\mathrm{tg}(w)$ for some w, and by recovering w, discarding the 10^* tail, and simulating the computation for L. By Lemma 5, $\mathrm{tg}(L') \leqslant_m^{\mathrm{PL}}$ GAP under a PL-covered reduction, and both are graph problems. By the main theorem of Section 3, there is a polynomial time m-reduction from the succinct version of the first to that of the second. So we have the following chain of reducibilities, which implies PSPACE-completeness by transitivity:

$$L \leqslant_m L' \leqslant_m s(\mathrm{tg}(L')) \leqslant_m s\mathrm{GAP}. \qquad \square$$

This is to say, the power of graph search algorithms, working on such implicit graphs given by expansion prodecures, captures exactly (functional and optimization versions of) PSPACE, and will only admit a polynomial time solution (in the size of the configurations) if P = PSPACE. Moreover, the space hierarchy theorem (see [1]) guarantees that *there is no subpolynomial memory space solution* for this problem. ("Subpolynomial" was defined in the preliminaries.) Of course, all algorithms used in practice are memory consuming in the worst case. PSPACE-completeness implies that there is no substantially cheaper alternative.

Note that this result holds for all algorithm schemes described in a generic way in terms of vertex expansions, including A* and other best-first search methods, among others. Of course, some particular search problems lie in complexity classes potentially below PSPACE, and particular algorithms for them may be more efficient. To bypass our hardness result, however, the price is to design specific algorithms extracting from their input more information than plainly a way of expanding vertices.

5. Some approximability issues

In many specific practical situations, the state space has weights in the edges, thus allowing for a definition of optimal solution, but only an approximate solution is actually needed. We argue here that the problem of finding a value which falls within any given constant factor of the optimal cost is as difficult, from the (worst-case) structural complexity point of view, as knowing whether there is a solution. So, no algorithm can be proved to furnish in polynomial time even an approximation, within a given interval, to the optimum cost, unless P = PSPACE.

Technically, the proof is not difficult. It relies on the most basic existing technique for proving such results: creating a gap in the costs. More precisely, a reduction is defined that, for each instance graph, creates a new graph in which there is always a solution path, having however a very high cost; substantially cheaper solutions are constructed on the basis of the solutions for the given instance. Hence, there is a solution for the given graph if and only if in the resulting construction there is a cheap path. Moreover, the difference is ample enough that a rough approximation to the optimal cost suffices to distinguish both cases. We claim that this construction can be performed as well in our succinct representation model.

Theorem 6. *The problem of approximating within any constant factor the cost of a solution path in a succinctly represented graph is* PSPACE-*hard.*

Proof. Fix the constant factor c. Given any circuit representing a graph of n vertices, convert it into another circuit acting as follows: on any nonsource vertex, output the original list of successors, associating unit cost to each edge; and on the source vertex, output the original list of successors extended with an additional new vertex, associating cost c^2n to this additional edge, and unit cost to all the other edges; and finally, on the new additional vertex, output as only successor the goal vertex, again with a cost of c^2n. Note that $O(\log n)$ bits (output wires) suffice to write down this cost. The size of the new circuit is therefore polynomial on the size of the original one.

Now it is easy to see the following property: there is a path from the source to the goal vertex in the graph represented by the given circuit if and only if, in the constructed circuit, any approximation to the optimal cost to a factor of c is less than cn. Indeed, if there is a path in the graph represented by the given circuit, since each edge is preserved with unit cost, there is a path of cost at most n, so approximating this value to factor c implies an approximate value of at most cn. On the other hand, if there is no such path in the given graph, then the only solution in the constructed graph has cost $2c^2n$, and any approximation to a factor of c of this value is at least $2cn$.

Thus, we have a polynomial time reduction from *s*GAP to the problem of approximating the cost of the optimal path in a graph given by a succinct representation, and it follows that the latter is PSPACE-hard. □

It is easy to see that the proof works as well for weaker approximations such as approximating to a polynomial, since a cost of n^c can still be output with $O(\log n)$ wires.

6. Application to AND/OR graphs

Another problem of wide practical interest is the search in AND/OR graphs, also called "problem reduction spaces" or also "alternating graphs".[1] In these, each nongoal vertex carries a label as "universal" or "existential". A solution is no longer a path, but a subgraph in which each existential vertex has exactly one successor and each universal vertex has all its successors; all the leaves must be goal nodes. A prime set of examples of application is given by games (although the problem appears in many other guises).

The AND/OR graph problem (AGAP, also standing for alternating GAP) is: given an AND/OR directed graph, a start vertex, and a set of goal vertices, decide whether there is a solution subgraph. The obvious weighted variant is also useful.

Again, algorithms (like AO*) exist for this problem, and various relevant algorithms are specifically tailored to games, in which the minimax principle is applied. Let us mention only alpha-beta pruning and SSS* [28]. The complexity of searching AND/OR graphs can be assessed from the abundant studies on the complexity of games. In particular, in [31] a number of games are proved to be EXPTIME-complete.

In that reference, a game is defined as given by the set of positions (encoded as words of a given length, and split into existential and universal) and a polynomial time algorithm to check validity of a move. There is a condition that "the board cannot be enlarged during the play", i.e., that valid moves are between positions encoded by words of the same length. Succinct representations as defined here (i.e., easy enough expansion algorithms) do not necessarily exist for the games from [31], and for some of them the (high) degree of each vertex makes them impossible. But, actually, careful examination proves that for some of the EXPTIME-complete games from [31], polynomial time expansion algorithms (i.e., succinct representations as defined here) do exist. In particular, the "game" on propositional formulas labeled G_2 in that paper has easy expansion algorithms, is EXPTIME-complete, and reduces to *s*AGAP (i.e., can be solved by AO*). A proof that *s*AGAP is EXPTIME-complete follows immediately from this fact.

Theorem 7. *sAGAP is* EXPTIME-*complete.*

This implies, by the time hierarchy theorem (see [1]), that no polynomial (nor even subexponential) time solution exists at all for this problem: EXPTIME-complete problems are provably intractable. The heuristic search methods such as AO* *are therefore unavoidable* to obtain solutions to practical cases in feasible computer time, and *will never be proved to provide a guarantee* of feasibility since such feasibility provably does not hold. (Note that no such provability can be stated for A* since it is possible, as of now, that P = PSPACE, or equivalently that polynomial time algorithms for *s*GAP exist.)

[1] The reason is that, without loss of generality, one can assume that through each path AND (universal) vertices and OR (existential) vertices alternate. The problem is often modeled equivalently with hypergraphs [29].

Let us comment briefly on an alternative proof, based on our succinct representations and essentially analogous to that of the case of GAP. We believe that this argumentation of the theorem is of independent interest. The problem GAME is proved P-complete in [17], under logspace reducibility. That problem is simply a rephrasing of AGAP in terms of players, where there are two sets of positions (the existential and universal vertices respectively), a starting position (source), a set of winning positions (goals), and a set of allowable moves (edges). The problem is to find out whether the starting position is winning for the first player (who plays on existential vertices). This is equivalent to finding out whether a solution subgraph exists.

The proof of P-completeness in [17] relies on a general reduction (through an intermediate problem) that is easily seen to be computable in polylog time on indexing machines, and PL-coverable, by the same argument as for GAP. Thus, we can refine the notion of reducibility (to the polylog time PL-covered case) for the P-completeness of GAME, and immediately we get EXPTIME-completeness of its succinct version, *s*AGAP, from our Theorem 1. We should point out here that concrete, specific, natural cases, such as appropriately formalized chess (and also checkers) problems, are known already to be EXPTIME-complete [14].

Note that in the definition of "game" in [31], it is assumed that the only knowledge readily available about the game is a polynomial time test for legitimacy of a move, corresponding closely to the previously used notion of succinct representations via circuits; more precisely, the condition that valid moves are between positions encoded by words of the same length allows for implementing the polynomial time algorithm as a circuit, and therefore the games from [31] correpond to a succinct representation in terms of transition matrices (in the sense of [8]) of the GAME problem of [17], and thus it follows from [2] that there are EXPTIME-complete games. Note, however, that in principle these facts do not speak of algorithms such as AO*, which are based on vertex expansions instead of recognizing edges. Indeed, since the practical algorithms assume more, namely, ability to expand a vertex, our more complex model of succinct representation is necessary to model search problems on implicit AND/OR graphs.

7. Discussion

The complexity of graph search problems, when the graph is given only implicitly as a procedure for vertex expansion (computing a list of successors) has been characterized as a function of the same problem on a standard representation such as the transition matrix.

A technical condition on the reductions (coverability) was necessary. It is interesting to note that several other reductions between (seemingly noncomplete) problems in nondeterministic LOGSPACE are not PL-coverable under our definition [18]; however, they become so under a somewhat more involved notion of coverability which takes into account the outdegree of the input graph. We have refrained from following this more general path since the simpler definition was sufficient for our purposes, but a theory of coverable reductions (which could conceivably have other interesting applications) should be based in a more general form, capturing as many existing reductions as possible [6,18].

Back to the study of the complexity of graph search problems under our succinct representation, we have argued that the relevance of this approach stems from the fact that many practical search algorithms can be seen as exploring an exponentially large graph for which a feasible vertex expansion procedure exists, such as A* or AO*. But there is a different context in which a similar consideration has been made: certain classes of total multivalued nondeterministically computable functions. These include PLS [15] and functions related to parity arguments [25].

All these classes contain relevant, practical problems, and sometimes they turn out to be complete for the class. For instance, a large number of heuristic algorithms for the traveling salesman problem consider tours or partial tours and "move" from one to another. Indeed, in [11] a number of heuristics for TSP are classified as "tour construction", where exploration "moves" among progressively more complete partial tours, or "tour improvement", where one tour is changed into another searching for better payoffs; mixed strategies are also considered.

In this sense, there is a huge graph, each of whose nodes is a (partial) tour, and whose edges indicate how to move from one to another. In this context, the input graph is nothing but a succinct representation of the graph of all its (partial) tours. Local search algorithms (such as those using the Lin-Kernighan-type heuristics [26]) are actually searching this large, succinctly represented graph. The problem to be solved (finding a vertex, i.e., a tour, below a given cost) lies in nondeterministic polylogarithmic time, and accordingly the succinct representation, TSP, is NP-complete in the decisional version.

Consider the following functional version based on a given local search algorithm: given an input, find the precise output of the local search started on it. This can be seen as the search for a path in the exponentially large graph defined by the neighborhood structure of the local search scheme; indeed, accordingly, for hard enough local search schemes this problem is PSPACE-hard [15,24]. However, other functional versions in which any local optimum suffices as output lie probably lower: in PLS [15]. The approach used here is, thus, consistent with the known results, and might be informative as of properties of these local search classes and problems. Similar considerations can be made with respect to the classes introduced in [25] based on "parity" or on "pigeonhole" arguments (PPA, PPAD, and PPP).

A major obstacle seems to be the fact that all these classes of functions below functional NP correspond to total functions, i.e., some sort of the so-called "promise problems" in which only inputs fulfilling certain "promise" are to be solved correctly. It is not clear to the author what could be the appropriate notion for obtaining from this intuition more general results about the classes PLS, PPA, PPAD, and PPP.

Acknowledgements

The following persons have been very helpful, either through personal discussions, proofreading, or both: Núria Castell, Josep Diaz, Ricard Gavaldà, Montse Hermo, Antoni Lozano, Pedro Meseguer, Jacobo Torán, and Carme Torras. Special thanks are due to

Georg Gottlob, for indicating that the scope of applications of the succinct representation technique was broader than I thought, and for suggesting the use of the (transitive) polylog time reducibility instead of log time reducibility. The help of the referees in improving the paper is also acknowledged.

References

[1] J.L. Balcázar, J. Diaz and J. Gabarró, *Structural Complexity I* (Springer, Berlin, 1988).
[2] J.L. Balcázar, A. Lozano and J. Torán, The complexity of algorithmic problems on succinct instances, in: R. Baeza-Yates and U. Manber, eds., *Computer Science: Research and Applications* (Plenum, New York, 1992).
[3] M. Cadoli, F.M. Donini, P. Liberatore and M. Schaerf, The size of a revised knowledge base, in: *Proceedings 14th Symposium on Principles of Database Systems* (1995) 151–162.
[4] M. Cadoli, T. Eiter and G. Gottlob, Default logic as a query language, in: *Proceedings 4th International Conference on Principles of Knowledge Representation and Reasoning* (1994).
[5] A.K. Chandra, D.C. Kozen and L.J. Stockmeyer, Alternation, *J. ACM* **28** (1981) 114–133.
[6] A.K. Chandra, L.J. Stockmeyer and U. Vishkin, Constant depth reducibility, *SIAM J. Comput.* **13** (1984) 423-439.
[7] T. Eiter, G. Gottlob and H. Mannila, Adding disjunction to DATALOG, in: *Proceedings ACM Symposium on Principles of Database Systems* (1994).
[8] H. Galperin and A. Wigderson, Succinct representations of graphs, *Inform. Control* **56** (1983) 183–198.
[9] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness* (Freeman, San Francisco, CA, 1979).
[10] G. Gogic, H. Kautz, C. Papadimitriou and B. Selman, The comparative linguistics of knowledge representation, in: *Proceedings IJCAI-95*, Montreal, Que. (1995) 862-869.
[11] B.L. Golden and W.R. Stewart, Empirical analysis of heuristics. in: E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan and D.B. Shmoys, eds., *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization* (Wiley, New York, 1985).
[12] R. Greenlaw, H.J. Hoover and W.L. Ruzzo, *A Compendium of Problems Complete for P* (Oxford University Press, Oxford, 1994).
[13] J.-W. Hong, On some deterministic space complexity problems, *SIAM J. Comput.* **11** (1982) 591-601.
[14] D.S. Johnson, A catalog of complexity classes, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity* (North-Holland, Amsterdam, 1990) 67-161.
[15] D.S. Johnson, C.H. Papadimitriou and M. Yannakakis, How easy is local search?, *J. Comput. Syst. Sci.* **37** (1988) 79-100.
[16] N.D. Jones, Space-bounded reducibility among combinatorial problems, *J. Comput. Syst. Sci.* **11** (1975) 68-75.
[17] N.D. Jones and W.T. Laaser, Complete problems for deterministic polynomial time, *Theoret. Comput. Sci.* **3** (1977) 105-117.
[18] N.D. Jones, Y.E. Lien and W.T. Laaser, New problems complete for nondeterministic log space, *Math. Syst. Theory* **10** (1976) 1-17.
[19] M. Kowaluk and K.W. Wagner, Vector language: simple description of hard instances, in: *Proceedings Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science **452** (Springer, Berlin, 1990) 378-384.
[20] S. Kundu, A new variant of the A* algorithm which closes a node at most once, *Ann. Math. Artif. Intell.* **4** (1991) 157-176.
[21] T. Lengauer and K.W. Wagner, The correlation between the complexities of the nonhierarchical and hierarchical versions of graph problems, *J. Comput. Syst. Sci.* **44** (1992) 63-93.
[22] K. Mehlhorn and S. Näher, LEDA: a platform for combinatorial and geometric computing, *Commun. ACM* **38** (1995) 96-102.
[23] C.H. Papadimitriou, The complexity of the Lin-Kernighan heuristic for the Traveling Salesman Problem, *SIAM J. Comput.* **21** (1992) 450-465.

[24] C.H. Papadimitriou, *Computational Complexity* (Addison-Wesley, Reading, MA, 1994).
[25] C.H. Papadimitriou, On the complexity of the parity argument and other inefficient proofs of existence, *J. Comput. Syst. Sci.* **48** (1994) 498-532.
[26] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity* (Prentice-Hall, Englewood Cliffs, NJ, 1982).
[27] C.H. Papadimitriou and M. Yannakakis, A note on succinct representations of graphs, *Inform. Control* **71** (1986) 181-185.
[28] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving* (Addison-Wesley, Reading, MA, 1984).
[29] E. Rich and K. Knight, *Artificial Intelligence* (McGraw-Hill, New York, 2nd ed., 1991).
[30] W.J. Savitch, Relations between nondeterministic and deterministic tape complexities, *J. Comput. Syst. Sci.* **4** (1970) 177-192.
[31] L.J. Stockmeyer and A.K. Chandra, Provably difficult combinatorial games, *SIAM J. Comput.* **8** (1979) 151-174.
[32] K.W. Wagner, The complexity of combinatorial problems with succinct input representation, *Acta Inform.* **23** (1986) 325-356.