# Efficiently Computing Join Orders with Heuristic Search

IMMANUEL HAFFNER, Saarland University, Saarland Informatics Campus, Germany
JENS DITTRICH, Saarland University, Saarland Informatics Campus, Germany

Join order optimization is one of the most fundamental problems in processing queries on relational data. It has been studied extensively for almost four decades now. Still, because of its NP hardness, no generally efficient solution exists and the problem remains an important topic of research. The scope of algorithms to compute join orders ranges from exhaustive enumeration, to combinatorics based on graph properties, to greedy search, to genetic algorithms, to recently investigated machine learning. A few works exist that use heuristic search to compute join orders. However, a theoretical argument why and how heuristic search is applicable to join order optimization is lacking.

In this work, we investigate join order optimization via heuristic search. In particular, we provide a strong theoretical framework, in which we reduce join order optimization to the shortest path problem. We then thoroughly analyze the properties of this problem and the applicability of heuristic search. We devise crucial optimizations to make heuristic search tractable. We implement join ordering via heuristic search in a real DBMS and conduct an extensive empirical study. Our findings show that for star- and clique-shaped queries, heuristic search finds optimal plans an order of magnitude faster than current state of the art. Our suboptimal solutions further extend the cost/time Pareto frontier.

CCS Concepts: • **Information systems** → **Query planning**; **Query optimization**.

Additional Key Words and Phrases: query optimization; query planning; join ordering

**ACM Reference Format:**
Immanuel Haffner and Jens Dittrich. 2023. Efficiently Computing Join Orders with Heuristic Search. *Proc. ACM Manag. Data* 1, 1, Article 73 (May 2023), 26 pages. https://doi.org/10.1145/3588927

## 1 INTRODUCTION

The *Structured Query Language* (SQL) is the dominant programming language to query and transform relational data, that is usually stored in *(relational) database management systems* ((R)DBMS). SQL is a declarative language: it only expresses *what* to compute without specifying *how* to compute. This declarative style of expressing operations burdens a DBMS with determining a query execution plan (or simply query plan) that defines how the computations required by a query are done. A crucial part of determining a query plan is determining a join order, i.e. the order in which individual relations are joined by the respective join predicates of the query. The join order has a major impact on the performance of the query plan and hence it is of utmost importance to a DBMS to compute a *"good"* join order – or at least to avoid *"bad"* join orders [1, 19]. This problem is known as the *join order optimization problem* (JOOP) and it is generally NP hard [3, 16]. There exists a comprehensive body of work on computing join orders. It can be divided into work on computing optimal join orders [3, 6, 11, 12, 16, 22, 32], work on greedy computation of potentially

Authors' addresses: Immanuel Haffner, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany, immanuel.haffner@bigdata.uni-saarland.de; Jens Dittrich, Saarland University, Saarland Informatics Campus, Saarbrücken, Germany, jens.dittrich@bigdata.uni-saarland.de.

2836-6573/2023/5-ART73
https://doi.org/10.1145/3588927

suboptimal join orders [9, 24, 25, 37], work on adaptive re-optimization of join orders [17, 26, 28, 38], and recent work based on machine learning [20, 21, 23].

Ono and Lohman [27] derive analytically the number of distinct plans w/o Cartesian products, showing that the amount of plans is generally exponential in the number of relations. For queries with many relations, the search space of plans quickly becomes too large to explore exhaustively. DBMSs therefore define a threshold beyond which suboptimal but faster algorithms are used [25]. Interestingly though, optimal algorithms need not be exhaustive.

In the domain of AI planning, searching extremely large search spaces is a frequent task and research in that area has brought forth algorithms to efficiently explore such search spaces. An important class of such algorithms is *best-first search* (BFS). BFS enables efficiently finding optimal or nearly optimal solutions without exhaustively exploring the entire search space. It has proven itself useful in a wide range of applications [8, 30]. The question arises whether and how BFS can be applied to JOOP.

### 1.1 Contributions

In this work, we present a new approach to join order optimization that is based on heuristic search, an important subset of BFS. In particular, we make the following contributions.

(1) To the best of our knowledge, we present the first formal reduction of JOOP to shortest path. We present formalizations for both bottom-up and top-down join ordering and investigate their dualism. (Section 2)
(2) We define heuristic search, perform a theoretical analysis of heuristic search applied to our shortest path problem, and elaborate the general search procedure. (Section 3)
(3) We present an efficient search space representation for both bottom-up and top-down search. Additionally, we devise two crucial optimizations, one of which is highly particular to the search space of JOOP. (Section 4)
(4) We identify and circumvent a potential pitfall when incorporating a DBMS cost model into heuristic search, that severely limits the efficiency of the search. (Section 5)
(5) We experimentally evaluate our approach and compare it to state-of-the-art algorithms. (Section 7)
(6) We propose a new benchmark that systematically explores the Query Graph Exploration Landscape (QGraEL) along the three query graph dimensions number of relations, graph density, and edge distribution. (Section 7.3)

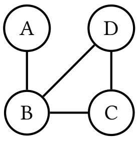
The paper is organized in the order of contributions. We discuss related work in Section 6.

## 2 JOIN ORDER OPTIMIZATION AS A SHORTEST PATH PROBLEM

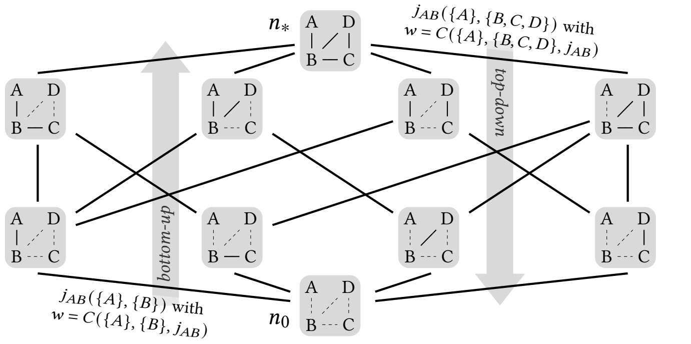
In this section, we formalize join order optimization as a shortest path problem. We begin with a brief excursion to shortest path and graph search. We then reduce bottom-up join order optimization to shortest path. We investigate the dualism of bottom-up and top-down join order optimization when expressed as a shortest path problem. Lastly, we analyze the time complexity of solving JOOP via shortest path.

### 2.1 The Shortest Path Problem

We formally define the shortest path problem on directed graphs as follows. Let $G := (V, E)$ be a graph with vertices $V$ and directed, weighted edges $E \subseteq \{(u, v, w) \mid u, v \in V, w \in \mathbb{R}^+\}$. For an edge $e = (u, v, w)$, we call $u$ the *tail*, $v$ the *head*, and $w$ the *weight* of $e$. A path $P = e_1 \dots e_k \in E^k$ is a sequence of edges with $\forall\, i \in \{1, \dots, k-1\}.\ \mathit{head}(e_i) = \mathit{tail}(e_{i+1})$. We say that $P$ *starts* in $\mathit{tail}(e_1)$, *ends* in $\mathit{head}(e_k)$, and has length $|P| = k$. The weight of a path is defined as



(a) Example query graph of four relations and four joins.


$n_*$
$j_{AB}(\{A\}, \{B, C, D\})$ with $w = C(\{A\}, \{B, C, D\}, j_{AB})$
top-down
bottom-up
$j_{AB}(\{A\}, \{B\})$ with $w = C(\{A\}, \{B\}, j_{AB})$
$n_0$

(b) Search space for the shortest path. Sets are represented by their induced subgraph, e.g. $n_0$ is $\{\{A\}, \{B\}, \{C\}, \{D\}\}$.

Fig. 1. Join ordering reduced to a shortest path problem.

$weight(P) \coloneqq \sum_{i=1}^{|P|} weight(e_i)$. Let $n_0, n_* \in V$ be the start and goal of a search problem, respectively, and further let $\mathbb{P}(n_0, n_*) \coloneqq \{P \mid P \text{ starts in } n_0 \land P \text{ ends in } n_*\}$ be the set of all paths from $n_0$ to $n_*$. We then define the shortest path as

$$\underset{P \in \mathbb{P}(n_0, n_*)}{\mathbf{arg\,min}}\, weight(P) \qquad \text{(Def. 1)}$$

A shortest path algorithm computes a solution for the shortest path problem, that is a shortest path algorithm computes for some $G \coloneqq (V, E)$ and $n_0, n_* \in V$ a path $P$ according to Def. 1.

## 2.2 Reducing JOOP to Shortest Path

We will now formulate join ordering as a shortest path problem. To do so, we need to formalize JOOP and then reduce it to shortest path. Note, that this requires a reduction from NP-hard JOOP to PTIME shortest path where the size of the search space for shortest path is exponential in the size of the query graph $G_Q$ (in the worst case). We use Figure 1 as a running example throughout this section. For some query $Q$, let $G_Q \coloneqq (R, J)$ be the *query graph* of $Q$, with relations $R$ as vertices and joins $J \subseteq \binom{R}{2}$[1] as edges. For Figure 1a, we have $R = \{A, B, C, D\}$ and $J = \{\{A, B\}, \{B, C\}, \{B, D\}, \{C, D\}\}$. The goal of join order optimization is to order the joins in $J$ such that the induced *plan* for $Q$ has minimal cost. In this work, we restrict ourselves to binary, inner joins. We call a subset $S \subseteq R$ a *subproblem* of $Q$, e.g. $\{A, B, D\}$ is one subproblem of $Q$. We say that a join $j = \{r_1, r_2\} \in J$ joins two disjoint subproblems $S_1$, $S_2$ if $r_1 \in S_1 \land r_2 \in S_2$. For example, join $j_{BD} = \{B, D\} \in J$ joins subproblems $\{A, B\}$ and $\{D\}$, written $j_{BD}(\{A, B\}, \{D\})$. Every join $j \in J$ can hence be treated as a partial function $2^R \times 2^R \rightharpoonup 2^R$:

$$j : (S_1, S_2) \mapsto S_1 \uplus S_2 \quad \text{if} \quad j = \{r_1, r_2\} \land r_1 \in S_1 \land r_2 \in S_2$$

The precondition of $j$ ensures that $j$ is only applicable to two disjoint subproblems that are joinable by $j$. For example, $j_{BD}(\{A, B\}, \{D\}) = \{A, B, D\}$ while $j_{BD}(\{A, B\}, \{C\})$ is undefined. Note, that this definition of $j$ requires specifying the two subproblems $S1, S2$ to join. However, we want to represent a query plan as a sequence of joins, i.e. without explicitly specifying for each join what subproblems are joined. We can leverage a join $j$'s precondition to formulate a *hoisted* definition $j^*$

[1]The notation $\binom{S}{k}$, read "from set $S$ choose $k$", denotes all subsets of $S$ of size $k$, i.e. $\{s \subseteq S \mid |s| = k\}$. Observe, that $\left|\binom{S}{k}\right| = \binom{|S|}{k}$.

that operates on sets of subproblems. In particular, $j^*$ automatically selects from a set of pairwise disjoint subproblems $S_1, \dots, S_n$ the two subproblems $S_i, S_k$ for which $j(S_i, S_k)$ is defined:

$$j^* : \{S_1, \dots, S_n\} \quad \mapsto \quad \Big(\{S_1, \dots, S_n\} \setminus \{S_i, S_k\}\Big) \cup \{j(S_i, S_k)\}$$

where $j = \{r_i, r_k\}$, $r_i \in S_i$, and $r_k \in S_k$. For example, $j^*_{BD}\big(\big\{\{A, B\}, \{C\}, \{D\}\big\}\big) = \big\{\{C\}\big\} \cup \big\{j_{BD}(\{A, B\}, \{D\})\big\}$. With the hoisted definition of joins, we can define a plan as a sequence of joins. Let $p = j_1 \dots j_n$ be a plan. Then $p(\mathcal{S}) \coloneqq j^*_n \circ \dots \circ j^*_1(\mathcal{S})$ denotes the sequential application of joins to some set of subproblems $\mathcal{S}$. Any plan $p$ joining all relations $R$ of query $Q$, formally $p\left(\binom{R}{1}\right) = \{R\}$, is a feasible plan for $Q$. Consider, for example, the plan $p = j_{BD} j_{AB} j_{CD}$. We have

$$\begin{aligned}
p(\binom{R}{1}) &= j^*_{CD} \circ j^*_{AB} \circ j^*_{BD}\big(\big\{\{A\}, \{B\}, \{C\}, \{D\}\big\}\big) \\
&= j^*_{CD} \circ j^*_{AB}\big(\big\{\{A\}, \{B, D\}, \{C\}\big\}\big) = j^*_{CD}\big(\big\{\{A, B, D\}, \{C\}\big\}\big) \\
&= \big\{\{A, B, C, D\}\big\} = \{R\}
\end{aligned}$$

Note, that $j_{BC}$ does not occur in the plan as it is subsumed by applying both $j_{BD}$ and $j_{CD}$. The goal of join order optimization is hence to compute a feasible plan $p$ of minimal cost.

We now reduce JOOP to shortest path. The idea of our reduction is that every application of a join $j$ to two subproblems $S_1, S_2$ forms an edge in the search space for shortest path. The weight of this edge is the cost of performing this join. Search starts in the initial vertex $n_0 = \binom{R}{1}$. The search space consists of all vertices reachable from $n_0$ through successive application of the joins in $J$. Figure 1b shows the search space for the query graph in Figure 1a. The search space is a directed graph, with edges directed *away* from $n_0$ (read bottom to top). The vertices of the search space are sets of subproblems yet to be joined together. Each subproblem of a vertex is drawn as a *connected subgraph* (csg) with solid edges; dashed edges represent joins not yet applied. Every path from the start $n_0 = \binom{R}{1}$ to the goal $n_* = \{R\}$ is a sequence of joins joining all relations in $R$ and therefore a feasible plan according to our definition. Further, the weight of such a path equals the cost of the corresponding plan. We call the search space constructed by this reduction of JOOP to shortest path $\text{SP}_{\text{JOOP}}$. We can now solve JOOP by computing a shortest path according to Def. 1 in $\text{SP}_{\text{JOOP}}$.

So far, we did not explain how weights are computed. It is fair to assume that a DBMS can provide a cost model to predict the cost of joining two subproblems $C : 2^R \times 2^R \times J \longrightarrow \mathbb{R}^+$. With cost model $C$, we can define the weight of an edge as

$$weight\big((u, v)\big) \coloneqq \min \left\{ C(S_1, S_2, j) \,\middle|\, \begin{array}{l} j \in J \land S_1, S_2 \in u \land \\ v = (u \setminus \{S_1, S_2\}) \cup \{j(S_1, S_2)\} \end{array} \right\}$$

A join subsumed by other joins can be evaluated in two ways: either by a join algorithm that supports a conjunction of multiple predicates or by a selection succeeding the subsuming joins. Either way, we expect $C$ to compute the costs accordingly.

## 2.3 The Dualism of Bottom-Up and Top-Down Join Order Optimization

The reduction in Section 2.2 is for *bottom-up* join ordering: initially all relations are disjoint in $n_0$ and then joins are applied to join subproblems until *all* relations are joined together in $n_*$. In *top-down* join ordering, we start with all relations already joined together and *"undo"* joins until all relations are pairwise disjoint. Undoing joins means partitioning a subproblem $S$ into smaller subproblems $S_1, S_2$ with $\exists\, j \in J.\ j(S_1, S_2) = S$. Observe in Figure 1b that *top-down* join ordering corresponds to a search starting in $n_*$ with goal $n_0$ and edges directed *towards* $n_0$. The search space of top-down join ordering is dual to that of bottom-up join ordering. Hence, top-down join ordering is the dual problem of bottom-up join ordering.

## 2.4 Complexity of $\text{SP}_{\text{JOOP}}$

Join order optimization is well-known to be NP hard [3, 16]. This means that solving JOOP requires time exponential in the size of the query graph $G_Q$. Since our reduction of JOOP to shortest path preserves optimality, solving JOOP by computing a shortest path in $\text{SP}_{\text{JOOP}}$, that is constructed by our *exponential-time* reduction in Section 2.2, must have worst-case time *exponential* in the size of the query graph. To prove that this is indeed the case, we give the following constructive argument.

Ono and Lohman [27] show that queries whose query graph $G_Q = (R, J)$ is a clique have $\Theta(3^{|R|})$ many *connected complement pairs* (ccp), where a ccp is a pair of subproblems $(S_1, S_2)$ s.t. $\exists\, j \in J.\ j(S_1, S_2)$ and $S_1, S_2$ induce csgs in $G_Q$. We show that for each ccp in $G_Q$ there exists *at least* one vertex in $\text{SP}_{\text{JOOP}}$ : For every ccp $(S_1, S_2)$ in $G_Q$ there exists at least one set of subproblems $\mathcal{S}$, s.t. $\mathcal{S}$ contains the ccp. Exactly one such $\mathcal{S}$ contains the ccp and otherwise only base relations, i.e. $\mathcal{S} = \{S_1, S_2\} \cup \binom{R \setminus (S_1 \cup S_2)}{1}$. This $\mathcal{S}$ is a vertex in $\text{SP}_{\text{JOOP}}$ . Hence, $\text{SP}_{\text{JOOP}}$ has $|V| \in \Omega(3^{|R|})$ many vertices. Because every vertex in $\text{SP}_{\text{JOOP}}$ (except the goal) has at least one outgoing edge, there are $|E| \in \Omega(3^{|R|})$ many edges.

For computing a shortest path, we can choose from a broad set of shortest path algorithms. Because we are only interested in shortest paths from $n_0$ to $n_*$, our problem is the special case *single-pair* shortest path, with pair $(n_0, n_*)$. Schrijver [31] gives an extensive survey of shortest path algorithms. In the class of *uninformed* (or *blind*) search, algorithms only have information of the start $n_0$ and the search space (cf. Figure 1b). This effectively means that the knowledge of $n_*$ is of no use to uninformed search. Of this class of algorithms, even the asymptotically best have a worst-case time complexity that is at least linear in the size of the search space, i.e. $\Omega(|V|)$ or $\Omega(|E|)$. Since both $|V|$ and $|E|$ of $\text{SP}_{\text{JOOP}}$ are exponential in the size of the query graph $G_Q$ (in the worst case), computing a shortest path in $\text{SP}_{\text{JOOP}}$ requires time exponential *in the size of* $G_Q$ (in the worst case). However, the mentioning of uninformed search suggests that there must be *informed* search. Informed search, or *heuristic* search, has additional knowledge beyond the problem description, that allows for a goal-oriented search. We discuss this in the following Section 3.

## 3 JOOP AS A HEURISTIC SEARCH PROBLEM

After reducing JOOP to $\text{SP}_{\text{JOOP}}$ in Section 2, we explore how to solve $\text{SP}_{\text{JOOP}}$ with heuristic search. We therefore extend search by a *heuristic function.* The heuristic function (or just heuristic) estimates for a given vertex in the search space the weight of a shortest path from that vertex to a goal. The heuristic enables the search to focus on vertices that it deems to lead to shorter paths. We can apply heuristic search to our shortest path problem if we can define a heuristic for our search space. We discuss important properties of heuristic functions in Section 3.1 and their impact on heuristic search in Section 3.2. We motivate that heuristic search enables us to gradually sacrifice optimality, in terms of plan cost, for efficiency. In Section 3.3, we then describe conceptually how we apply heuristic search to $\text{SP}_{\text{JOOP}}$ with a potentially exponentially large search space. We present proof sketches for completeness, soundness, and optimality in Section 3.4 and study different performance criteria of heuristic search in Section 3.5. We discuss how to find a heuristic for $\text{SP}_{\text{JOOP}}$ in Section 5.

### 3.1 Properties of Heuristic Functions

A heuristic function $h$ estimates for some vertex $v$ the weight of a shortest path from $v$ to a goal. The optimal heuristic $h^*$ returns for each vertex $v$ *exactly* the weight of a shortest path from $v$ to a goal. A heuristic $h$ is *goal-aware* if the heuristic value of any goal is 0, formally $v \textit{ is goal} \Rightarrow h(v) = 0$. In $\text{SP}_{\text{JOOP}}$, checking whether a vertex is a goal is simple and we therefore assume all heuristics to be goal-aware. A heuristic *underestimates* if there exists a vertex for which the heuristic underestimates the weight of a shortest path to goal, i.e. $\exists\, v \in V.\ h(v) < h^*(v)$. Likewise, a heuristic *overestimates* if $\exists\, v \in V.\; h(v) > h^*(v)$. A heuristic that *never overestimates* is called *admissible.* Admissibility becomes important when we discuss optimality of heuristic search. A heuristic $h$ is called *consistent* if the heuristic never overestimates the weight of a single edge, i.e. $\forall\, (u, v, w) \in E.\; h(u) \leq h(v) + w$. Every consistent and goal-aware heuristic is also admissible and $h^*$ is consistent.

## 3.2 Properties of Heuristic Search

To exploit a heuristic we need to perform heuristic search. In particular, we will focus on Dijkstra's algorithm [7] and famous $A^*$ [15]. There are two interesting properties of $A^*$, that we will mention here, as they will guide us when we design and evaluate heuristics.

**Optimality.** Algorithm $A^*$ is optimal, that is it computes the shortest path from start to goal, if the heuristic $h$ is admissible [15].
**Time Complexity.** Dechter and Pearl [5] have shown that if the heuristic $h$ is consistent, algorithm $A^*$ is optimally efficient, i.e., there exists no BFS algorithm that finds a shortest path with traversing fewer vertices of the search space.

According to these two properties, if we are able to devise an admissible heuristic for $\mathrm{SP}_{\mathrm{JOOP}}$, we are guaranteed that $A^*$ will find a shortest path, which corresponds to an optimal plan of JOOP. Further, if we are able to devise a consistent heuristic for $\mathrm{SP}_{\mathrm{JOOP}}$ that is efficiently computable, i.e. in PTIME, we know that we can efficiently solve $\mathrm{SP}_{\mathrm{JOOP}}$ (even if not in PTIME). A naïve attempt would be to devise an optimal heuristic for $\mathrm{SP}_{\mathrm{JOOP}}$. However, an optimal heuristic for $\mathrm{SP}_{\mathrm{JOOP}}$ cannot be computed in PTIME:

**Theorem 1.** Unless P = NP, any optimal heuristic for $\mathrm{SP}_{\mathrm{JOOP}}$ is not in PTIME, formally: $\forall\, h.\; \big(\forall\, v.\; h(v) = h^*(v)\big) \Rightarrow h \notin \mathrm{PTIME}$.

We will prove Theorem 1 by contradiction, showing that if an optimal heuristic $h$ were computable in PTIME, we could solve JOOP in PTIME, contradicting the fact that JOOP is NP hard [3, 16]. Our proof relies on the conjecture P $\neq$ NP and on bounding the complexity of shortest path. With a depth $d$, where $d$ is the minimal length of any path from $n_0$ to $n_*$, and a maximum branching factor $b$, the complexity of shortest path is in $O(b^d)$ [29]. In $\mathrm{SP}_{\mathrm{JOOP}}$, $b$ is bounded by the number of joins $|J|$ and $d$ is exactly $|R| - 1$. Hence, we can bound the time complexity by $O(|J|^{|R|-1})$. So far, this bound is not really helpful. However, for optimal $h$, $b$ becomes 1. We prove this by contradiction (compare [8, Theorem 2.9 on p. 72]):

**Lemma 1.** Assume an edge $(u, v, w) \in E$ and further $\forall\, (u, v', w') \in E.\; h^*(v) + w \leq h^*(v') + w'$. Then $v$ lies on a shortest path from $u$ to a goal.

Proof of Lemma 1 by contradiction. Assume $v$ does not lie on a shortest path from $u$ to goal. Then $\exists\, (u, v', w') \in E$ with $v \neq v'$ and $v'$ lies on a shortest path from $u$ to goal. By optimality of $h^*$, it holds $h^*(u) = h^*(v') + w'$. Because $v$ does not lie on a shortest path from $u$ to goal, it holds $h^*(u) < h^*(v) + w$. Hence, $h^*(v') + w' < h^*(v) + w$. $\lightning$ $\square$

By Lemma 1, if $h$ is optimal, it is sufficient for a search algorithm to pursue only a single edge minimizing $h(v) + w$. This means, with an optimal heuristic the branching factor $b$ becomes 1 and our time complexity bound collapses to $O(1^{|R|-1}) = O(1)$. However, our bound does not account for the evaluation of $h$.

Proof of Theorem 1 by contradiction. With $d = |R| - 1$ and $b \leq |J|$, $h$ is evaluated at most $\big(|R| - 1\big) \cdot |J|$ times. This term is polynomial in the size of the query graph $G_Q = (R, J)$. Assume for the sake of contradiction that we have an optimal heuristic $h \in \mathrm{PTIME}$. We would then be able to

**Algorithm 1** BFS with on-demand search space computation.

```
1:  function BFS($n_0$ : start vertex)
2:      $\mathcal{L} \leftarrow [n_0]$                                    ▷ initialize open list
3:      while $\mathcal{L}$ not empty do
4:          $u, g_u \leftarrow$ EXTRACT-BEST($\mathcal{L}$)             ▷ extract next best vertex with its cost
5:          if $u$ is goal then
6:              return Success                                ▷ found path from $n_0$ to $n_*$
7:          end if
8:          for each $(u, v, w) \in E$ in EXPAND($u$) do                ▷ expand $u$
9:              ADD($\mathcal{L}, v, g_u + w$)                        ▷ add successors of $u$ to $\mathcal{L}$
10:         end for
11:     end while
12:     return Failure                                        ▷ no path from $n_0$ to $n_*$ was found
13: end function
```

compute a shortest path in $\mathrm{SP}_{\mathrm{JOOP}}$ in PTIME. Since a shortest path in $\mathrm{SP}_{\mathrm{JOOP}}$ is an optimal plan for JOOP, we would be able to solve JOOP in PTIME. $\lightning$ □

We now know that an optimal heuristic cannot be computed efficiently, i.e. in PTIME. However, if we were able to approximate $h^*$ by some heuristic $h \in$ PTIME and $h$ were consistent, then we could compute an optimal solution optimally efficiently with $A^*$. Interestingly, for practical purposes, the heuristic need not be consistent to achieve an effective branching factor close to 1.

## 3.3 Searching an Exponentially Large Space

In Section 2.4 we learned that the number of vertices $|V|$ and the number of edges $|E|$ of $\mathrm{SP}_{\mathrm{JOOP}}$ are exponential in the size of the query graph $G_Q$. Hence, finding a shortest path in $\mathrm{SP}_{\mathrm{JOOP}}$ requires time exponential *in the size of* $G_Q$. Constructing the *entire* search space a priori to the actual search would render our approach inefficient if not infeasible. Therefore, in our algorithm we will explore the search space on demand only.

We describe conceptually how the search space is computed successively and on demand by BFS and provide pseudo-code in Algorithm 1. We assume that BFS uses a container of vertices, usually called *open list*; some BFS algorithms implement this open list as a priority queue (e.g. $A^*$), some implement it as a single vertex (e.g. hill climbing). In each iteration of BFS, the *best* vertex is extracted from the open list to be *expanded* next (line 4). The definition of best depends on the search algorithm and is usually based on some combination of the weight $g$ of the path by which the vertex was reached from the start $n_0$ and the heuristic value $h$ of the vertex. For example, $A^*$ defines best as the vertex minimizing $g + h$. When a vertex is expanded, some or all successors (but at least one) of this vertex are computed and added to the open list (line 9). BFS proceeds until one of two termination criteria is met: (1) When a goal is extracted from the open list (line 5), BFS has found a path from start to this goal and the search terminates successfully. (2) When the open list runs empty, no path from start to a goal was found and BFS terminates unsuccessfully (line 12).

Depending on the BFS algorithm, we may in the worst case explore the entire search space before reaching a goal. For $A^*$, this may happen when all joins have nearly the same cost, making edge weights nearly uniform and degrading $A^*$ to breadth-first search. We believe, that situations in which (almost) the entire search space is explored are highly unlikely in the real world. We provide Table 1 in Section 3.5, which supports our claim with empirical data.

Figure 2 shows an example run of $A^*$ on the search space in Figure 1b. The algorithm starts in vertex $n_0$ = ①. We annotate each vertex with the weight $g$ of the path from start ①. Additionally, we annotate vertices with their heuristic value $h$. For example, start ① has $g = 0, h = 50$. Vertex ① represents the four subproblems $\{A\}$, $\{B\}$, $\{C\}$, and $\{D\}$. We expand ① by applying any one of the four possible joins in $J$, thereby generating four successors. For each such successor we compute the weight $g$ of the path from the start to that successor as well as the heuristic $h$. We label each edge with its weight, as resulting from expansion. For the expansion of ①, edges are labeled 20,

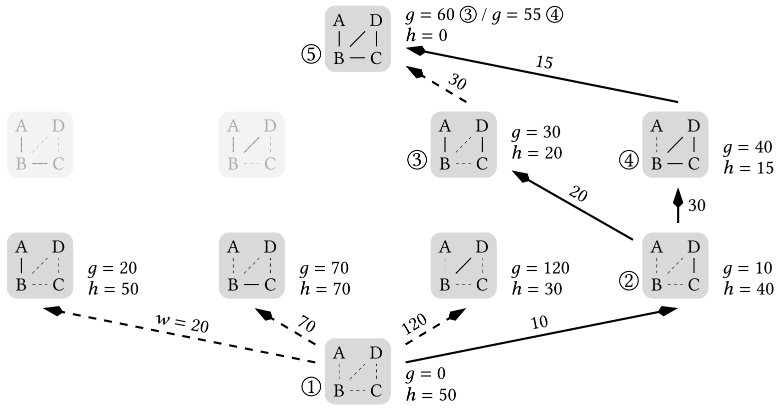


Fig. 2. Search tree for bottom-up $A^*$. Vertices are labeled ① with their order of expansion. Dashed edges mark vertices generated by expansion and are labeled with the weight of the edge. Solid edges additionally were pursued by the search. Two vertices are never generated and the goal is generated twice, first with $g = 60$ by ③ and then with $g = 55$ by ④. The final plan is $A \bowtie (B \bowtie (C \bowtie D))$.

70, 120, and 10. The vertex of minimal $g + h$ that was not yet expanded is expanded next. Here, ② with $g + h = 10 + 40 = 50$ is expanded. ② represents the three subproblems $\{A\}$, $\{B\}$, and $\{C, D\}$. From ②, we can either join $\{A\}$ with $\{B\}$ or $\{B\}$ with $\{C, D\}$. Hence, the two successors ③ & ④ are generated by expansion of ②. We again compute $g$ and $h$ of these successors. Next, ③ with $g + h = 50$ is expanded into goal ⑤ with $g = 60, h = 0$. Notice, that at this point, the algorithm does *not* terminate yet, as we only *added* a goal ⑤ (line 9 in Algorithm 1). The next vertex to expand is ④ and expansion yields ⑤, again. However, this time ⑤ is reached by a path with weight $g = 55$. The next vertex to expand is ⑤ with $g + h = 55 + 0$. Since ⑤ is a goal, the search terminates successfully with a shortest path of weight 55.

## 3.4 Completeness, Soundness, Optimality

In this section, we show that the successive computation of the search space during BFS does not harm completeness, soundness, and optimality. We therefore sketch the proofs for these properties, assuming a BFS algorithm that is complete, sound, and optimal. However, before we do so, let us highlight three properties of the search space, that will help us with the proofs. (1) The search space is *acyclic*: there is no non-empty path that starts and ends in the same vertex. (2) The search space has *no dead-ends*: every vertex except the goal has at least one successor. (3) The *goal $n_*$ has* depth $|R| - 1$: every path from $n_0$ to $n_*$ has *exactly* length $|R| - 1$. These properties follow directly from the construction of the search space in Section 2.2.

**Completeness.** As the search space has no dead-ends, EXPAND yields for each non-goal vertex at least one successor. Because the search space is acyclic, each EXPAND reduces the distance (in edges) to $n_*$ by one. Therefore and because the goal has finite depth, BFS will find a path from $n_0$ to $n_*$, if one exists.

**Soundness.** Soundness of our heuristic search follows from soundness of the reduction in Section 2.2, soundness of the BFS algorithm, and soundness of EXPAND. The latter is sound if it yields for a given vertex $u$ only successors $v$ w.r.t. $E$, i.e. $\exists\,(u, v, w) \in E$.

**Optimality.** In Section 2.2 we have shown that an optimal solution of the problem reduced to shortest path corresponds to an optimal solution of JOOP. Assuming optimality of BFS, what remains to show is that EXPAND preserves optimality. This is the case if for every vertex $u$, EXPAND$(u)$ yields

Table 1. Comparison of $\text{DP}_{\text{CCP}}$ to search with Dijkstra and $A^*_{\downarrow} + h_{sum}$ by #ccps enumerated to compute an optimal plan.

| | chain | | | cycle | | | star | | | clique | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 5 | 10 | 15 | 5 | 10 | 15 | 5 | 10 | 15 | 5 | 10 | 15 |
| $\text{DP}_{\text{CCP}}$ | 20 | 165 | 560 | 36 | 321 | 1106 | 32 | 2304 | 114 688 | 90 | 28 501 | 7 141 686 |
| Dijkstra↑ | 10 | 438 | 2702 | 13 | 130 | 2875 | 18 | 218 | 404 | 22 | 2884 | 26 992 |
| Dijkstra↓ | 20 | 202 | 2629 | 44 | 650 | 6288 | 24 | 1026 | 61 739 | 91 | 191 313 | >30 000 000 |
| $A^*_{\downarrow} + h_{sum}$ | 10 | 53 | 522 | 16 | 118 | 557 | 10 | 265 | 12 568 | 29 | 20 969 | 7 050 206 |

at least one vertex $v$ s.t. $v$ lies on a shortest path from $u$ to goal. If EXPAND yields all successors, this is trivially the case.

We now have a profound understanding of heuristic search and its applicability to our shortest path problem. Before we dive into the algorithmic challenges in Section 4, we present different performance criteria for evaluating heuristic search.

## 3.5 Performance Criteria

There are different measures to assess the performance of heuristic search. Of course, we look at running times in our evaluation in Section 7. If we allow for suboptimal solutions, an additional measure is how far off a computed solution is from an optimal solution (i.e. a shortest path). Another measure, that we already learned about in Section 3.2, is the effective branching factor $b^*$, which allows us to evaluate how informative a heuristic is to the search. However, $b^*$ cannot be measured directly but is derived from the depth of the goal (which is $|R| - 1$) and another important measure for heuristic search: the number of generated vertices [29]. Because the depth of the goal is fixed, the only remaining variable for computing $b^*$ is the number of generated vertices. Therefore, this number alone already allows us to compare different heuristics by how informative they are to the search. It also allows us to compare different search algorithms by how goal-oriented they explore the search space. We can even compare our approach to classical dynamic programming (DP): the number of vertices generated corresponds to the number of ccps joined and can directly be compared to the number of ccps enumerated by DP.

With this knowledge, we conduct our first experiment. We compare classical DP implemented by $\text{DP}_{\text{CCP}}$ [22] to blind and heuristic search. As blind search we perform Dijkstra's algorithm in both directions: bottom-up, labeled Dijkstra↑, and top-down, labeled Dijkstra↓. As heuristic search we perform $A^*_{\downarrow}$ (top-down) with admissible heuristic $h_{sum}$ (cf. Section 5). We compare the three algorithms by the number of ccps joined in Table 1. Because heuristic $h_{sum}$ is admissible, $A^*_{\downarrow}$ computes an optimal plan, like $\text{DP}_{\text{CCP}}$ and Dijkstra's algorithm. The best result in each column is underlined. Before we draw conclusions from our experiment, we want to emphasize that $\text{DP}_{\text{CCP}}$ enumerates *all* ccps *exactly once* without duplicates [22]. The proportion of unique ccps to the number of relations $|R|$ is polynomial for chain and cycle topologies and exponential for star and clique topologies [22, 27]. From the results in Table 1, we can make two key observations: (1) On star and clique topologies, Dijkstra↑ enumerates significantly less ccps than $\text{DP}_{\text{CCP}}$. On chain and cycle topologies, $A^*_{\downarrow} + h_{sum}$ enumerates significantly less ccps than $\text{DP}_{\text{CCP}}$. We conclude that heuristic search is able to find an optimal plan without enumerating *all* ccps; sometimes only a fraction of all ccps is required. (2) When we compare our two top-down searches, Dijkstra↓ and $A^*_{\downarrow} + h_{sum}$, we observe how much of an impact the heuristic has on the search: the heuristic sometimes reduces the number of ccps enumerated by more than one order of magnitude.

According to Observation 1, search and particularly heuristic search finds a provably optimal plan without the need to enumerate *all* ccps. In contrast, traditional algorithms for computing an optimal solution enumerate all ccps (with or without duplicates) [11, 12, 22, 32, 35, 36]. Whether it is possible to compute an optimal plan via search in less time depends on whether search can be implemented efficiently. We describe the algorithmic challenges we face in the following Section 4. Observation 2 exemplifies the impact the heuristic has on the search's performance. Therefore, in Section 5, we explore and evaluate different heuristics.

## 4 ALGORITHMIC CHALLENGES

To be able to efficiently search for a shortest path from $n_0$ to $n_*$, we must be able to efficiently explore the search space. In Section 3.3, we argue that we must not compute the entire search space a priori to the search but instead compute the explored regions successively. Exploring the search space is done by successively expanding vertices to their successors, as exemplified in Figure 2.

In Section 4.1, we present a vertex representation that enables efficient generation of successors via EXPAND and efficient evaluation of a heuristic function $h$. Consequently, in Section 4.2 we show how to efficiently compute successor vertices for this representation in bottom-up and top-down search. As the search's performance also heavily depends on the implementation of the open list, we present in Section 4.3 an implementation that supports fast insertion of generated successors, fast extraction of the next best vertex, and efficient handling of duplicates. As we shall see in Section 4.4, some duplicates are actually desired while others are undesired. We develop an algorithm to suppress the generation of undesired duplicates already when expanding a vertex, thereby preventing attempts to insert undesired duplicates into the open list.

### 4.1 Vertex Representation

The vertices of our search space are sets of subproblems, i.e. sets of sets of relations. We can incrementally assign to each relation in the query graph a unique index, starting at 1. For the query graph in Figure 1a, we could assign indices $A \mapsto 1$, $B \mapsto 2$, $C \mapsto 3$, $D \mapsto 4$. Each subproblem can then be represented as a bit vector $b_1 \dots b_{|R|}$ with bit $b_i$ set if relation with index $i$ is within the subproblem. For example, the subproblem $\{C, D\}$ is represented by the bit vector `0011`. A vertex is then represented as a sequence of bit vectors $\mathcal{V}$, with one bit vector per subproblem. Additionally, the bit vectors are kept sorted lexicographically to allow for hashing and efficient equality testing. For example, the vertex with label ② in Figure 2, $\big\{\{A\},\{B\},\{C,D\}\big\}$, is represented by $\mathcal{V} =$ [`1000`, `0100`, `0011`]. To efficiently store and operate on bit sets, we employ the same case distinction as Neumann and Radke [25], using a 64 bit integer for queries with up to 64 relations, a 128 bit integer for up to 128 relations, a dynamic array of 64 bit vectors for up to 1024 relations, and a dynamic array of sorted relations for more than 1024 relations.

### 4.2 Vertex Expansion

Given a vertex $u$, EXPAND$(u)$ must compute all outgoing edges of that vertex. The term "outgoing" is now relative to the direction of join ordering: outgoing edges in bottom-up join ordering are incoming edges in top-down join ordering and vice versa, cf. Figure 1b. Given the representation $\mathcal{V}_u$ of a vertex $u$ in the search space, the task is to compute all edges $(u, v, w) \in E$, i.e. the outgoing edges of $u$. We now make a case distinction about the search direction.

**Bottom-Up Search.** In bottom-up search, there exists an edge $(u, v, w) \in E$ if there is a join $j \in J$ s.t. $j^*(u) = v$. Expanding the definition of hoisted joins from Section 2.2, we get

$$j^*(u) = v \quad \Leftrightarrow \quad \exists\, S_1 \neq S_2 \in u.\, v = \big(u \setminus \{S_1, S_2\}\big) \cup \big\{j(S_1, S_2)\big\}$$

From this definition we can derive Algorithm 2 to compute all outgoing edges of $u$: we simply need to test for all pairs of subproblems $(S_1, S_2)$ whether there exists a join $j \in J$ s.t. $j(S_1, S_2)$.

*Time Complexity.* Algorithm 2 iterates over all pairs of subproblems (skipping symmetric pairs) and over all joins. Note that there can be at most $|R|$ many subproblems in $\mathcal{V}_u$. In the innermost

**Algorithm 2** Bottom-up vertex expansion.

```
function EXPANDBottomUp($\mathcal{V}_u$ : representation of vertex $u$)
    for $i = 1$ to $|\mathcal{V}_u| - 1$ do
        for $k = i + 1$ to $|\mathcal{V}_u|$ do
            $\overline{b_i} \leftarrow \mathcal{V}_u[i]$                                      ▷representation of subproblem $S_1$
            $\overline{b_k} \leftarrow \mathcal{V}_u[k]$                                      ▷representation of subproblem $S_2$
            for each $j \in J$ joining $\overline{b_i}$ and $\overline{b_k}$ do
                $l \leftarrow \mathcal{V}_u[1 : i-1] \circ \mathcal{V}_u[i+1 : k-1]$                ▷slice $\mathcal{V}_u$ to replace $\overline{b_i}$ ...
                $r \leftarrow \mathcal{V}_u[k+1 : |\mathcal{V}_u|]$                              ▷... and $\overline{b_k}$ by $(\overline{b_i} \mid \overline{b_k})$ ...
                $\mathcal{V}_v \leftarrow l \circ \left(\overline{b_i} \mid \overline{b_k}\right) \circ r$                        ▷... and maintain lexicographical order
                yield $\mathcal{V}_v$ by $j(S_1 = \overline{b_i}, S_2 = \overline{b_k})$                 ▷emit how to generate successor
            end for
        end for
    end for
end function
```

**Algorithm 3** Top-down vertex expansion.

```
function EXPANDTopDown($\mathcal{V}_u$ : representation of vertex $u$)
    for $i = 1$ to $|\mathcal{V}_u|$ do                                              ▷partition each subproblem of $\mathcal{V}_u$
        for each ccp $\left(\overline{b_1}, \overline{b_2}\right)$ in PARTITION$_{\text{MinCutAGaT}}$($\mathcal{V}_u[i]$) do       ▷[12, Fig. 6]
            $\mathcal{V}_v \leftarrow \mathcal{V}_u[1 : i-1] \circ \mathcal{V}_u[i+1 : |\mathcal{V}_u|]$                ▷remove subproblem at index i
            InsertSortedLex($\mathcal{V}_v$, $\overline{b_1}$)                                  ▷insert $\overline{b_1}$ lexicographically
            InsertSortedLex($\mathcal{V}_v$, $\overline{b_2}$)                                  ▷insert $\overline{b_2}$ lexicographically
            yield $\mathcal{V}_v$ by $j(S_1 = \overline{b_1}, S_2 = \overline{b_2})$                 ▷emit how to generate successor
        end for
    end for
end function
```

loop, $\mathcal{V}_u$ is sliced to construct $\mathcal{V}_v$. This can be done in a single pass over $\mathcal{V}_u$. We can therefore bound Algorithm 2's time by $O(|R|^3 \cdot |J|)$. Judging by the asymptotic runtime, our algorithm appears to be very inefficient. However, our experiments in Section 7 reveal that expansion makes up for only a small fraction of overall search time.

**Top-Down Search.** Analogously to bottom-up search, in top-down search, there exists an edge $(u, v, w) \in E$ if there is a join $j \in J$ s.t. $j^*(v) = u$. Again, by expanding the definition of hoisted joins from Section 2.2, we get

$$\begin{aligned} j^*(v) = u \quad &\Leftrightarrow \quad \exists\, S_1 \neq S_2 \in v.\ u = v \setminus \{S_1, S_2\} \cup \left\{j(S_1, S_2)\right\} \\ &\Leftrightarrow \quad \exists\, S_1 \neq S_2 \in v\ \exists\, S \in u.\ j(S_1, S_2) = S \end{aligned}$$

This means, there exists an edge $(u, v, w) \in E$ if there is a subproblem $S$ in $u$ that can be partitioned into subproblems $S_1, S_2$ in $v$ such that there is a join $j \in J$ with $j(S_1, S_2) = S$. Enumerating these partitions is exactly the problem of top-down join ordering [6, 11, 12, 36]. We select an existing partitioning algorithm to enumerate all ccps of a subproblem, here PARTITION$_{\text{MinCutAGaT}}$ [12], to implement top-down vertex expansion in Algorithm 3.

*Time Complexity.* Fender and Moerkotte [12] analyze the time complexity of MincutLazy and find that it is worst for clique queries with $O(|R|^2)$ time. MinCutAGaT, which is based on MincutLazy, exhibits the same asymptotic runtime behavior. Analyzing our Algorithm 3, we see that the outer-most loop performs no more than $|R|$ iterations. Further, slicing $\mathcal{V}_u$ to construct $\mathcal{V}_v$ and the two invocations of InsertSortedLex require at most $O(|R|)$ time. We conclude that Algorithm 3's time complexity is bounded by $O(|R|^4)$. Note that MinCutBranch [11] exhibits better asymptotic runtime behavior than MinCutAGaT for cycle and clique queries but is significantly more complex. As our evaluation in Section 7 shows, vertex expansion takes only a small share of overall search time and we therefore opt for MinCutAGaT in this work.

**Algorithm 4** Add vertex to open list with duplicate handling.

```
function ADD($\mathcal{L}$ : open list, $v$ : vertex to add, $g_v$ : cost of $v$)
    if $v$ in $\mathcal{L}$.table then                                              ▹ is $v$ a duplicate?
        if $g_v < \mathcal{L}$.table$[v].g$ then                                      ▹ reached $v$ on a cheaper path?
            if $\mathcal{L}$.table$[v]$.handle is None then                              ▹ not in open list?
                $\mathcal{L}$.table$[v]$.handle ← INSERT($\mathcal{L}$.heap, $v$, $g_v + h(v)$)
                $\mathcal{L}$.table$[v].g \leftarrow g_v$                                    ▹ remember cost of $v$
            else
                DECREASE-KEY($\mathcal{L}$.heap, $\mathcal{L}$.table$[v]$.handle, $g_v + h(v)$)   ▹ update cost
                $\mathcal{L}$.table$[v].g \leftarrow g_v$                                    ▹ remember updated cost
            end if
        end if
    else
        $\mathcal{L}$.table$[v]$.handle ← INSERT($\mathcal{L}$.heap, $v$, $g_v + h(v)$)           ▹ insert and save handle
        $\mathcal{L}$.table$[v].g \leftarrow g_v$                                            ▹ remember cost of $v$
    end if
end function
```

**Algorithm 5** Extract best vertex from open list.

```
function EXTRACT-BEST($\mathcal{L}$ : open list)
    $v, g_v$ ← FIND-MIN($\mathcal{L}$.heap)                                           ▹ get best vertex and its cost
    DELETE-MIN($\mathcal{L}$.heap)                                                   ▹ remove best vertex from heap
    $\mathcal{L}$.table$[v]$.handle ← None                                             ▹ update handle
    return $v, g_v$
end function
```

## 4.3 Open List and Duplicate Detection

As shown in Algorithm 1, BFS extracts in each iteration of the outer loop the next best vertex from the open list with EXTRACT-BEST, expands it into its successors, and adds the successors to the open list with ADD. To efficiently implement EXTRACT-BEST and ADD, we require a data structure that efficiently supports (1) finding the next best vertex, (2) removing the next best vertex, and (3) adding newly generated successor vertices. There is one more operation that the data structure should support. To motivate this, let us look again at the example in Figure 2. The goal ⑤ is generated twice, first by ③ with $g = 60$ and afterwards by ④ with $g = 55$. When ⑤ is generated the second time, it is already present in the open list. One way to support this, is by allowing for duplicates in the open list. This is safe, since duplicates have the same heuristic value $h$ and hence the duplicate with smaller $g$ is extracted first from the open list. However, if duplicates occur frequently, they cause the open list to grow unnecessarily large, thereby degrading performance. A better way to cope with duplicates is to detect them while they are being added to the open list. We therefore devise a scheme for the *detection of duplicates* (DEDUP): A *new* vertex is immediately added to the open list. When a *duplicate* vertex is being added to the open list, we compare the $g$ values of this duplicate and the vertex already in the open list. A duplicate with equal or greater $g$ is discarded, as it cannot lead to finding shorter paths. A duplicate with smaller $g$ means that we have found a shorter path from $n_0$ to this vertex. Instead of adding the duplicate to the open list, we reduce $g$ of the vertex *already within the open list* to $g$ of the duplicate that is being added. The data structure should therefore also support an operation to (4) reduce $g$ of an already incorporated vertex.

Data structures fit for this task are heaps. They provide exactly the aforementioned required operations (1) FIND-MIN, (2) DELETE-MIN, (3) INSERT, and (4) DECREASE-KEY. There are many different implementations of heaps, e.g. binary heap, binomial heap, Fibonacci heap, and pairing heap, to name a few [4, 6.1 Heaps on p. 151]. Some heaps do not support DECREASE-KEY. In that case, the operation can be emulated by first deleting the vertex of old $g$ and then re-inserting the vertex with new $g$. However, we shall use `boost::heap::fibonacci_heap`, which efficiently supports the DECREASE-KEY operation.[2]

[2]Note that BOOST implements max-heaps.

Table 2. The impact of DeDup. We run Dijkstra↑ on queries of 10 relations and count new and duplicate vertices.

| | | chain | cycle | star | clique |
|---|---|---|---|---|---|
| without DeDup | #new | 497 | 819 | 115 | 1497 |
| | #duplicates | 182 577 | 224 726 | 72 | 13 710 |
| with DeDup | #new | 497 | 819 | 115 | 1497 |
| | #duplicates | <u>1140</u> | <u>1408</u> | <u>22</u> | <u>191</u> |

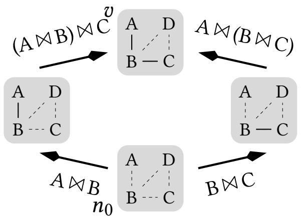
DeDup requires that we can search for a particular vertex in the heap – an operation that is usually not (efficiently) supported. We therefore naïvely use a hash table in addition to the heap. The hash table serves two purposes: (1) It stores for each seen vertex a handle. If the vertex is currently in the heap, the handle references the heap entry. Otherwise, the vertex has been extracted from the heap and the handle is **None**. (2) The hash table stores for each seen vertex the weight $g$ of the cheapest path by which the vertex was reached. With the handle we are able to perform a DECREASE-KEY operation when a vertex is reached on a cheaper path. It also enables us to identify whether a vertex is currently in the heap or has already been deleted. Storing $g$ inside the hash table enables us to discard duplicates not reached on a cheaper path, even when the vertex has already been extracted from the open list. DeDup implicitly requires that the heap can provide handles to entries. This is usually the case when the heap provides *referential stability*[3] of its elements. We implement ADD and EXTRACT-BEST with DeDup in Algorithm 4 and Algorithm 5, respectively. Note, that in the pseudo-code the open list $\mathcal{L}$ contains both the heap and the hash table. Further, Algorithm 4 defines the *best* vertex as the one minimizing $g_v + h(v)$, as is required by $A^*$; any other definition of *best* is possible.

We demonstrate the necessity of DeDup with a small experiment. We compare two implementations of the open list: one implementation with DeDup and one implementation that simply allows for duplicates and performs no duplicate checking at all. We compute optimal plans for queries of 10 relations using bottom-up search with Dijkstra's algorithm and we count the new and duplicate vertices generated by vertex expansion. We present our findings in Table 2. Note, that whether we allow for duplicates in the open list only affects how many duplicates are generated and how often the same vertex is expanded but it does not affect which *unique* vertices are expanded or generated. The explored region of the search space remains the same. Hence, and as we expect, the amount of newly generated, unique vertices is independent of whether we allow for duplicates. In stark contrast, the amount of duplicates generated when allowing for duplicate vertices in the open list is up to two orders of magnitude larger. To understand why allowing for duplicates in the open list has such a devastating effect on the amount of vertices generated, we have to understand that every single duplicate extracted from the open list is expanded and hence all its generated successors become duplicates in the open list, leading to an exponential blow-up of duplicates. In the following Section 4.4, we discuss how we further prevent some duplicates of ever being generated.

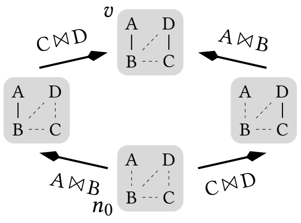
## 4.4 Duplicate Prevention

In previous Section 4.3, we proposed DeDup to efficiently eliminate duplicates in the open list. However, DeDup does not prevent the *generation* of duplicates during vertex expansion. In the following, we identify two classes of duplicates: desired and undesired duplicates. We then devise a scheme to *prevent the generation of undesired duplicates* during vertex expansion (PreDup). We consequently extend vertex representation from Section 4.1 and expansion from Section 4.2.

[3]Referential stability is also called *pointer stability.*



(a) Desired duplicate, with each of the two paths corresponding to a unique partial plan.



(b) Undesired duplicate, with both paths corresponding to the same partial plan.

Fig. 3. Example of desired versus undesired duplicates.

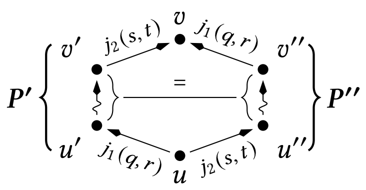


Fig. 4. General pattern of undesired duplicates. Vertex $u$ must contain four subproblems $q, r, s, t$, s.t. $q$ can be joined with $r$ and $s$ can be joined with $t$. The order of the two joins can be permuted, resulting in two paths $P'$ and $P''$ of exact same weight, i.e. $weight(P') = weight(P'')$. Hence, $v$ is generated twice with the same cost.

To introduce the notion of desired and undesired duplicates, let us consider the two examples in Figure 3. Both examples show a fraction of the search space of Figure 1b. In Figure 3a, we see two paths leading from the start $n_0$ to a vertex $v$, where relations A, B, and C have been joined. These two paths, despite leading to the same vertex, correspond to *two different partial plans*: one plan joins A and B first, the other joins B and C first. In contrast, in Figure 3b, we see two distinct paths from $n_0$ to $v$ that correspond to *the exact same partial plan*: although the two paths order the joins A ⋈ B and C ⋈ D differently, this ordering has no semantics in the corresponding partial plan. In the example of Figure 3a, we do want to generate the duplicate of $v$, as it may reveal a shorter path to $v$. If the duplicate does not reveal a shorter path, it will be discarded by DeDup (cf. Section 4.3). In the example of Figure 3b, however, we would be wise not to generate the duplicate of $v$. Its path corresponds to an already considered partial plan. Therefore, $v$ has already been generated with the exact same cost $g$ and hence the duplicate of $v$ will definitely be discarded by DeDup.

We devise a scheme to prevent the generation of such undesired duplicates during vertex expansion, named PreDup. We say that a duplicate vertex is undesired, if the vertex was reached before on some path $P'$ and is now reached on a different path $P''$ and it can be shown that – independent of cardinalities – $weight(P') = weight(P'')$. In that case, we say path $P''$ is *redundant*. We provide a general pattern of undesired duplicates in Figure 4. There is a redundant path between vertices $u$ and $v$, if $u$ has four subproblems $q, r, s, t$ such that $q$ can be joined with $r$ and $s$ can be joined with $t$, i.e. $\exists\, j_1, j_2 \in J.\ j_1(q,r), j_2(s,t)$. In that case, any path from $u$ to $v$ that includes both these joins can be transformed into another, valid path from $u$ to $v$ by exchanging the order of the two joins. These two paths have the exact same weight, hence $v$ is generated twice with exact same cost $g$. The idea of PreDup is to prevent the generation of undesired duplicates by preventing the search from pursuing redundant paths. More precisely, in Figure 4, the search may *either* perform $j_2(s,t)$ after $j_1(q,r)$ *or* $j_1(q,r)$ after $j_2(s,t)$ but not both. To implement this, we exploit the fact that our vertex representation in Section 4.1 keeps the sequence of bit vectors $\mathcal{V}$ sorted lexicographically.

**Algorithm 6** Extension of Algorithm 2 to prevent redundant paths.

```
function EXPANDBottomUp($\mathcal{V}_u$ : representation of vertex $u$)
    for $i = 1$ to $|\mathcal{V}_u| - 1$ do
        for $k = i + 1$ to $|\mathcal{V}_u|$ do
            $\overline{b_i} \leftarrow \mathcal{V}_u[i]$                          ▹representation of subproblem $S_1$
            $\overline{b_k} \leftarrow \mathcal{V}_u[k]$                          ▹representation of subproblem $S_2$
            if $\left(\overline{b_i} \mid \overline{b_k}\right) <_{\text{lex}} \nabla(\mathcal{V}_u)$ then      ▹undesired duplicate?
                continue                                                            ▹skip
            end if
            for each $j \in J$ joining $\overline{b_i}$ and $\overline{b_k}$ do
                $l \leftarrow \mathcal{V}_u[1 : i-1] \circ \mathcal{V}_u[i+1 : k-1]$
                $r \leftarrow \mathcal{V}_u[k+1 : |\mathcal{V}_u|]$
                $\mathcal{V}_v = l \circ \left(\overline{b_i} \mid \overline{b_k}\right)^{\nabla} \circ r$       ▹maintain lexicographical order
                yield $\mathcal{V}_v$ by $j(S_1 = \overline{b_i}, S_2 = \overline{b_k})$     ▹emit how to generate successor
            end for
        end for
    end for
end function
```

We have $j_1(q,r) = q \cup r$ and $j_2(s,t) = s \cup t$, with *either* $q \cup r <_{\text{lex}} s \cup t$ *or* $s \cup t <_{\text{lex}} q \cup r$. We store in $\mathcal{V}$ the subproblem that was the result of the most recent join, indicated with $\nabla$. By storing the most recently joined subproblem, we can suppress the generation of undesired duplicates during EXPAND: we skip joins whose join result is lexicographically smaller than the stored subproblem of the expanded vertex. For example, let $q <_{\text{lex}} r <_{\text{lex}} s <_{\text{lex}} t$. When expanding $u$ we get $u'$ with $\nabla(\mathcal{V}_{u'}) = q \cup r$ and $u''$ with $\nabla(\mathcal{V}_{u''}) = s \cup t$ and

$$\begin{array}{lllll} \text{for } \mathcal{V}_{u'} : & j_2(s,t) = s \cup t & \not<_{\text{lex}} & q \cup r = \nabla(\mathcal{V}_{u'}) & ✓ \\ \text{for } \mathcal{V}_{u''} : & j_1(q,r) = q \cup r & <_{\text{lex}} & s \cup t = \nabla(\mathcal{V}_{u''}) & ✗ \end{array}$$

Because we suppress join results that are lexicographically smaller than the stored subproblem, during vertex expansion the stored subproblem can only become larger w.r.t. the lexicographical ordering. Hence, no matter how $v''$ is reached from $u''$ in Figure 4, we know that $\nabla(\mathcal{V}_{v''}) \geq_{\text{lex}} \nabla(\mathcal{V}_{u''})$ and therefore $j_1(q,r)$ is suppressed when expanding $v''$. We extend our algorithm for bottom-up vertex expansion of Section 4.1 accordingly in Algorithm 6 and highlight the necessary modifications. In the initial vertex $n_0 = \binom{R}{1}$, the lexicographically smallest subproblem is marked. An extension of top-down expansion, as in Algorithm 3, would be analogous: the most recently partitioned subproblem is stored and subproblems lexicographically smaller than the most recently partitioned subproblem are not further partitioned. In the initial vertex of top-down search, i.e. $n_0 = \{R\}$, the single subproblem $R$ is marked. We need to show that BFS with EXPAND as in Algorithm 6 is still complete, sound, and optimal.

**Completeness.** Looking at Figure 4, we see that our scheme only prevents expanding $v''$ to $v$, i.e. the edge $v'' \rightarrow v$ , still leaving an alternative path from $u$ to $v$. More generally, all vertices that were reachable from $n_0$ before our modification are still reachable from $n_0$. This particularly holds true for $n_*$.

**Soundness.** Our scheme neither introduces new edges into the search space nor does it alter the weights of existing edges. Therefore, any path found from $n_0$ to $n_*$ corresponds to a feasible plan.

**Optimality.** In Figure 4, the paths $P'$ and $P''$ have the exact same weight. If either of the paths, say $P''$, is eliminated by our scheme, then $v$ is still reached by $P'$ of exact same weight as $P''$. If $P''$ was an optimal path, so is $P'$, and hence *an* optimal path to $v$ is found.

To evaluate the gain of PREDUP, we rerun the same experiment as in Table 2. This time, we use DEDUP (cf. Section 4.3) and compare bottom-up search with and without PREDUP. We present our findings in Table 3. Let us first look at star topology: We see that the number of generated new

Table 3. The impact of PreDup on the experiment of Table 2. Both configurations include DeDup (cf. Section 4.3).

| | | chain | cycle | star | clique |
|---|---|---|---|---|---|
| DeDup only | #new | 497 | 819 | 115 | 1497 |
| | #duplicates | 1140 | 1408 | 22 | 191 |
| DeDup + PreDup | #new | <u>377</u> | <u>626</u> | 115 | <u>891</u> |
| | #duplicates | <u>299</u> | <u>419</u> | 22 | <u>109</u> |

Table 4. The impact of the definition of edge weights on the number of vertices generated. We run Dijkstra↑ on queries of 10 relations. Both approaches compute a solution that is optimal w.r.t. Def. 2.

| | | chain | cycle | star | clique |
|---|---|---|---|---|---|
| #vertices generated | with Def. 3 | 1014 | 2484 | 344 | 1864 |
| | with Def. 4 | <u>676</u> | <u>1045</u> | <u>137</u> | <u>1000</u> |

and duplicate vertices does not change. This is expected, as in star topology there are no bushy plans and therefore there are no redundant paths and no undesired duplicates. Next we look at chain, cycle, and clique topologies: We see that the number of generated duplicates is significantly reduced, as we expected. However, and maybe to your surprise, we can also see that the number of generated new vertices shrunk. To understand why that is the case, let us reconsider Figure 2. With our scheme for preventing undesired duplicates, we have $\nabla(②) = \{C, D\}$. When ② is expanded, our scheme prevents the generation of ③ as successor, since $\{A, B\} <_{\text{lex}} \{C, D\}$. However, successor ④ is still generated and consequently a shortest path is found, with one unique vertex less generated.

## 5 HEURISTIC FUNCTIONS FOR JOOP

So far, we presented how to solve join order optimization by heuristic search and the algorithmic challenges that arose. In this section, we present three heuristics for the search problem and their respective heuristic properties. However, we must first understand how to derive edge weights from a DBMS' cost model and how heuristics depend on the cost model.

### 5.1 From DBMS Cost Model to Edge Weights

In Section 2.2 we introduced a cost model $C$ and used it to define the weights of the edges of the search space. The weight of a path is defined as the sum of the weights of its edges. A heuristic estimates the weight of a shortest path to the nearest goal. Therefore, heuristics depend on the cost model. It is hard, if not infeasible, to define an informative heuristic independent of the cost model. In this work, we focus on the well-known and frequently used cost model $C_{out}$, that assesses a plan by the sum of the cardinalities of all intermediate results [3, 11, 12, 24, 25]. It is recursively defined as

$$C_{out}(T) := \begin{cases} 0 & \text{if } T \in R \\ |T| + C_{out}(T_1) + C_{out}(T_2) & \text{if } T = T_1 \bowtie T_2 \end{cases} \tag{Def. 2}$$

In order to weigh the edges of the search space according to $C_{out}$, we must recursively decompose Def. 2 to match our definition of cost model $C$ from Section 2.2:

$$C(S_1, S_2, j) := |S_1 \bowtie S_2| \tag{Def. 3}$$

While this definition of $C$ is coherent with $C_{out}$, it has one major pitfall that significantly hurts bottom-up heuristic search: According to Def. 2, the cardinality of the result set of the query is always included in the total cost. When comparing entire plans by their cost, the cardinality of the result set always cancels out. This is, however, not the case when comparing two entries of the open list in bottom-up heuristic search: Every entry for the goal already includes in its $g$ the cardinality of the result set while entries for non-goal vertices do not include this cardinality in their $g$ yet, despite the fact that this cardinality occurs as edge weight on *any* path to goal. When the heuristic frequently underestimates, this leads to goals added to the open list being artificially pushed towards the end, delaying their expansion and ultimately delaying finding a plan. We therefore devise a variant of $C$ that simply excludes the cardinality of the result set:

$$C(S_1, S_2, j) := \begin{cases} 0 & \text{if } j(S_1, S_2) = R \\ |S_1 \bowtie S_2| & \text{otherwise} \end{cases} \tag{Def. 4}$$

Plans optimal w.r.t. Def. 4 are also optimal w.r.t. $C_{out}$ of Def. 2. We evaluate the impact of the two definitions of $C$ on bottom-up heuristic search by comparing the number of vertices generated in Table 4. Our experiment demonstrates that with Def. 4 bottom-up heuristic search converges faster towards a goal. Note that this pitfall does not exist in top-down heuristic search, as the cardinality of the result set is immediately incorporated in *all* vertices when the initial vertex is expanded.

## 5.2 Four Simple Heuristics

The following heuristics are designed particularly for $C_{out}$ of Def. 2, with edge weights computed according to Def. 4. In addition to the cost model, heuristics depend on the direction of the search, i.e. bottom-up vs. top-down, because heuristics estimate the distance to goal and the goal depends on the search's direction.

**The *zero* heuristic.** The simplest heuristic is the one assigning the same constant value to any vertex. This heuristic provides no additional information to the search. We define the particular constant heuristic with constant zero $h_{zero}(v) = 0$. Observe, that $h_{zero}$ is goal-aware, consistent, and admissible. When used as heuristic for $A^*$, the search degrades into Dijkstra's algorithm. Naturally, $h_{zero}$ can be used for both bottom-up and top-down search.

**The *sum* heuristic.** This heuristic provides a lower bound for the remaining cost to reach the goal in top-down search. All subproblems that are not base relations are yet to be partitioned. Looking at Def. 4, each subproblem that is not a base relation will add its cardinality to the overall cost $C_{out}$. We can therefore calculate a lower bound for the remaining cost by summing up the cardinalities of all subproblems that are no base relations:

$$h_{sum}(v) := \sum_{S \in v \setminus R} |S|$$

Since $h_{sum}$ is a lower bound of the remaining cost, it never overestimates and therefore it is admissible and can be used to compute an optimal plan. Note that $h_{sum}$ only accounts for the *current* subproblems, i.e. the subproblems in $v$. It does not consider any subproblems formed by partitioning subproblems in $v$. Therefore, $h_{sum}$ often underestimates the remaining cost dramatically and the error grows with the distance (in #edges) of $v$ to the goal.

**The *GOO* heuristic(s).** We devise a heuristic $h_{GOO\uparrow}$ for bottom-up search by greedily computing a *reasonable* path from the current vertex to goal using *greedy operator ordering* (GOO) [9]. GOO iteratively selects and joins two subproblems until only a single subproblem remains. The two subproblems to join $S_1, S_2$ are chosen to minimize $|S_1 \bowtie S_2|$. As $h_{GOO\uparrow}$ estimates the remaining distance to goal by computing an actual path, the heuristic never underestimates. However, because the subproblems to join $S_1, S_2$ are chosen greedily, the heuristic often overestimates. Hence, BFS with $h_{GOO\uparrow}$ does not guarantee finding an optimal plan. We also devise a variant of this heuristic for top-down search, named $h_{GOO\downarrow}$. This variant partitions each subproblem $S = S_1 \bowtie S_2$ s.t. $|S_1| + |S_2|$ is minimized.

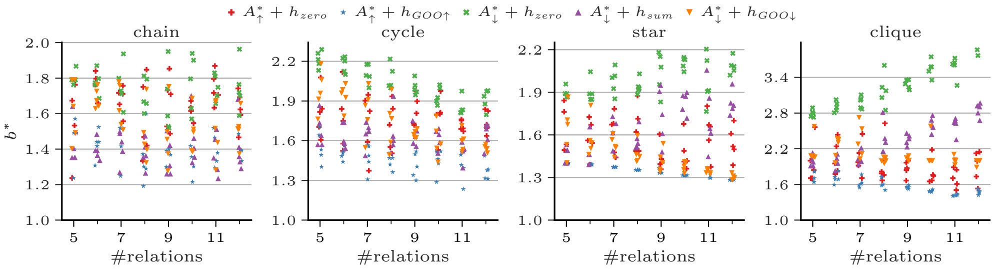


Fig. 5. Information value of different heuristics.

## 5.3 Informative Value of Heuristic Functions

We compare the four heuristics $h_{zero}$, $h_{sum}$, $h_{GOO\downarrow}$, and $h_{GOO\uparrow}$ by how informative they are to heuristic search. We assess the heuristics by their effective branching factor $b^*$ (cf. Section 3.2). Note, that $A^*$ with $h_{zero}$ is exactly Dijkstra's algorithm. We calculate $b^*$ – the *average* branching factor *per* vertex expansion – from the depth $d$ of the goal and the number $N$ of generated vertices by solving the following equation [29].

$$N = b^* + (b^*)^2 + \cdots + (b^*)^d \tag{Def. 5}$$

We experimentally determine $b^*$ for the four heuristics on the four topologies chain, cycle, star, and clique. We consider both bottom-up and top-down search with $A^*$. We vary the number of relations from 5 to 15, count the generated vertices, and from that derive the actual branching factor $b^*$ according to Def. 5. We repeat each experiment five times with different seed (for details see Section 7). We present our results in Figure 5. We can generally observe that different heuristics provide different information value to the search and that their information value varies between the query topologies. In particular, we make five important observations: (1) $A^*_\uparrow + h_{zero}$ is generally more informative than $A^*_\downarrow + h_{zero}$. This is due to Def. 4, where in bottom-up search the cost $g$ of a vertex already includes the cardinality of the current subproblems, which is not the case in top-down search. (2) $h_{sum}$ corrects the aforementioned deficiency of $h_{zero}$ and significantly reduces $b^*$. (3) For chain and cycle topology, $A^*_\downarrow + h_{sum}$ results in smaller $b^*$ than $A^*_\uparrow + h_{zero}$, while for star and clique topology, exactly the opposite is the case. This observation suggests that different types of queries are better solved by bottom-up or top-down search. (4) Both $A^*_\uparrow + h_{GOO\uparrow}$ and $A^*_\downarrow + h_{GOO\downarrow}$ achieve least $b^*$ for cycle, star, and clique topologies. This inadmissible heuristic causes the search to quickly converge towards a goal, but the solution can be suboptimal. (5) We can observe for each heuristic how $b^*$ evolves with growing number of relations. When $b^*$ grows with increasing number of relations, then the information value of the heuristic shrinks. Contrary, if $b^*$ shrinks with increasing number of relations, the information value of the heuristic grows. For example, $h_{GOO\downarrow}$'s information value for star topology grows the more relations the query involves. The information value of $h_{sum}$ for clique topology shrinks with increasing number of relations.

## 6 RELATED WORK

**Classical Join Ordering —** Ibaraki and Kameda [16] prove that the problem of join order optimization is generally NP hard, even when allowing for only a single join method (i.e. nested-loops join). The authors provide a polynomial-time greedy algorithm, that computes an optimal plan if the query graph is a tree, e.g. a star query. The algorithm requires that the cost function, under which optimization is performed, satisfies the *adjacent sequence interchange* (ASI) property. The ASI property requires that a cost-benefit ratio, named *rank*, can be computed for each join. The work was further extended by Krishnamurthy et al. [18] and the algorithm is sometimes referred to as IK/KBZ. Cluet and Moerkotte [3] show that summarizing the cardinalities of intermediate results serves as a good cost model, named $C_{out}$, that also satisfies the ASI property. We do not require ASI for heuristic search.

Selinger et al. [32] were the first to use DP to compute an optimal join order. Their algorithm, frequently referred to as $\text{DP}_{\text{size}}$, enumerates all (partial) plans in increasing number of relations, until a final, optimal plan is found. Cartesian products are performed as late as possible, i.e. never when the query graph is connected. Ono and Lohman [27] derive analytically for different topologies the number of distinct plans, excluding Cartesian products. For both star and clique topology, the number of plans is exponential in the number of relations. Vance and Maier [36] and Vance [35] improve upon $\text{DP}_{\text{size}}$ by devising a more efficient enumeration scheme following Gray code order [13]. This algorithm is frequently referred to as $\text{DP}_{\text{sub}}$. Moerkotte and Neumann [22] further improve plan enumeration via DP. Their algorithm $\text{DP}_{\text{CCP}}$ enumerates all connected pairs of connected subgraphs without duplicates by traversing the query graph in a particular order. Chaudhuri et al. [2] invent top-down plan enumeration by decomposing a set of relations into two smaller sets and recursively computing optimal plans for these sets. In their work, the authors only consider linear plans. DeHaan and Tompa [6] generalize top-down plan enumeration to bushy plans and exclude Cartesian products. Their algorithm builds upon efficiently finding minimal graph cuts by computing the biconnected components of the query graph, that are organized in the *biconnection tree*. The authors show that top-down planning integrates well with cost-based branch-and-bound pruning, however the benefit is limited when Cartesian products are excluded. Fender and Moerkotte [11, 12] further improve top-down plan enumeration with two algorithms: (1) $\text{TD}_{\text{MinCutAGaT}}$ extends the work of DeHaan and Tompa [6] by replacing the biconnection tree with an *advanced generate and test* routine. (2) $\text{TD}_{\text{MinCutBranch}}$ avoids connectedness checks by ensuring that only ccps are generated, thereby improving the complexity of finding a cut.

While the aforementioned algorithms enumerate *all* ccps, heuristic search is often able to find a *provably optimal* plan without enumerating all ccps. On the contrary, when the heuristic is uninformative, duplicates occur frequently. Branch-and-bound pruning is implicitly performed by the open list when ordering by $g$ or $g + h$ and $h$ is goal-aware. In contrast to prior work, heuristic search pursues those joins *first* that it deems to lead to cheaper plans.

**Greedy Join Ordering —** Fegaras [9] presents *greedy operator ordering* (GOO), a greedy algorithm that repeatedly joins in each iteration the two subproblems leading to the smallest result size, until all relations are joined. GOO is a BFS with the heuristic defined as the result size of the most recent join and greedy BFS as search.

Neumann [24] proposes query simplification to reduce the complexity of plan enumeration until it becomes tractable with DP. Simplification introduces ordering constraints, reducing the considered plans but sacrificing optimality. Neumann and Radke [25] propose *linearization* of the query graph. Their algorithm, named LinearizedDP, precedes $\text{DP}_{\text{CCP}}$ with a linearization phase based on IK/KBZ. Linearization, similarly to query simplification, reduces the amount of plans considered by DP, rendering DP suboptimal.

**Heuristic Search —** To the best of our knowledge, Sellis [33] work on MQO is the first to apply heuristic search in the context of query optimization. In this particular work, multiple plans are generated for each query and a heuristic search algorithm then selects for each query exactly one plan, considering common intermediate results and minimizing the overall cost of executing all queries. Note, that this is a different optimization problem than join order optimization, where a single plan for a single query is computed.

Marcus et al. [21] train an ML model to predict the cost of the best plan constructible from a given partial plan. They use this model as heuristic for BFS. An argument is missing as to why the problem can be solved by search and whether the learned model has good heuristic properties. A

**Algorithm 7** Generation of cardinalities.

```
function CARDINALITY-GEN($G_Q$ : query graph of query $Q$, $c_{min}$ : minimum cardinality, $c_{max}$ : maximum cardinality)
    $C \leftarrow$ new HashMap()                                   ▷ cardinalities of subproblems
    for each $r$ in $G_Q.R$ do                                     ▷ initialize cardinalities of base relations
        $C[r] \leftarrow$ RAND($c_{min}$, $c_{max}$)               ▷ random cardinality in range $c_{min}$ to $c_{max}$
    end for
    $C' \leftarrow$ new HashMap()                                  ▷ maximum possible cardinality per subproblem
    for each csg $(S_1, S_2)$ of $G_Q$ do                          ▷ enumerated in $DP_{CCP}$ [22] order
        $c_1 \leftarrow$ CARDINALITY$(S_1, C, C', c_{min}, c_{max})$   ▷ get cardinality of $S_1$
        $c_2 \leftarrow$ CARDINALITY$(S_2, C, C', c_{min}, c_{max})$   ▷ get cardinality of $S_2$
        if $S_1 \cup S_2$ not in $C'$ then
            $C'[S_1 \cup S_2] \leftarrow c_1 \cdot c_2$             ▷ set max. cardinality of $S_1 \cup S_2$
        else
            $C'[S_1 \cup S_2] \leftarrow$ MIN$(C'[S_1 \cup S_2], c_1 \cdot c_2)$   ▷ update max. cardinality
        end if
    end for
    $C[G_Q.R] \leftarrow$ CARDINALITY$(G_Q.R)$                     ▷ cardinality of result
    return $C$
end function

function CARDINALITY$(S, C, C', c_{min}, c_{max})$
    if $S$ not in $C$ then                                         ▷ no fixed cardinality for $S$ yet?
        $c' \leftarrow$ MIN$(C'[S], {c_{max}}^2)$                  ▷ max. cardinality of $S$, bounded by ${c_{max}}^2$
        $C[S] \leftarrow c_{min} + (c' - c_{min}) \cdot$ RAND$(0, 1)$   ▷ random cardinality of $S$
    end if
    return $C[S]$
end function
```

general analysis of the search problem is lacking. Since the learned model may overestimate plan costs, the heuristic is inadmissible and hence search is suboptimal.

## 7 EVALUATION

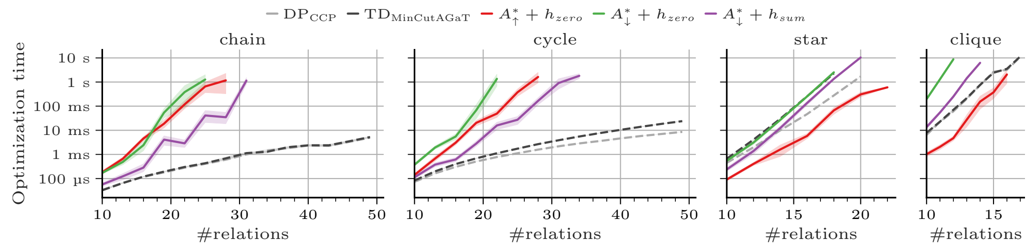
### 7.1 Setup

**System —** We implement our heuristic search and related state-of-the-art join ordering algorithms in mu*t*able [14], a main-memory database system currently developed at our group. Queries are provided to mu*t*able as SQL statements, for which mu*t*able computes a query plan with one of the join ordering algorithms. We use cost model $C_{out}$ in all experiments. mu*t*able provides an interface to read cardinalities from a file. We use this feature to provide exact cardinalities to the process of join order optimization. We further exploit this feature to simulate queries with varying selection and join selectivities without the need to generate actual data.

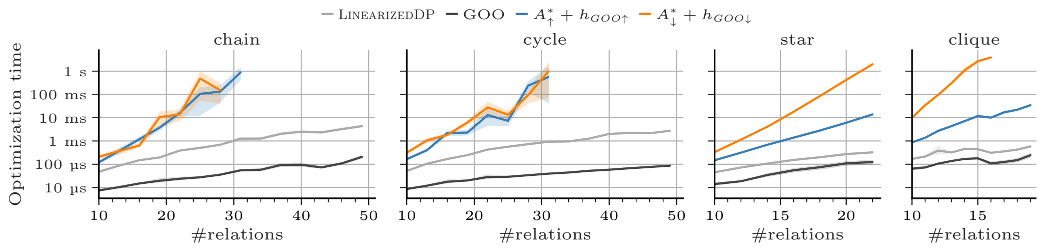
**Data —** We evaluate all algorithms on the four query topologies chain, cycle, star, and clique, as they are a de-facto standard for evaluating join order optimization [6, 10, 12, 22, 24, 34]. In addition, with this work we introduce a new benchmark which includes the former four topologies as special cases (Section 7.3). We vary number of relations and to simulate varying selection and join selectivities, we randomly generate 10 cardinality files per query. A file assigns to each subproblem a cardinality, where the cardinality assigned to a base relation represents the cardinality after selection (i.e. including selection selectivities) and the cardinality assigned to a subproblem of multiple relations represents the cardinality after selection *and* join (i.e. including selection and join selectivities). We randomly generate these cardinalities with our algorithm CARDINALITY-GEN, given in Algorithm 7. Note that our algorithm produces *correlated* selectivities, i.e. it does not hold in general that $sel(A \bowtie B \bowtie C) = sel(A \bowtie B) \cdot sel(B \bowtie C)$.

**Hardware —** We run our experiments on a desktop computer with an AMD Ryzen Threadripper 1900X CPU at 3.8 GHz and 32 GiB DDR4 main memory. We disable the CPU's dynamic frequency scaling to reduce noise in our measurements.

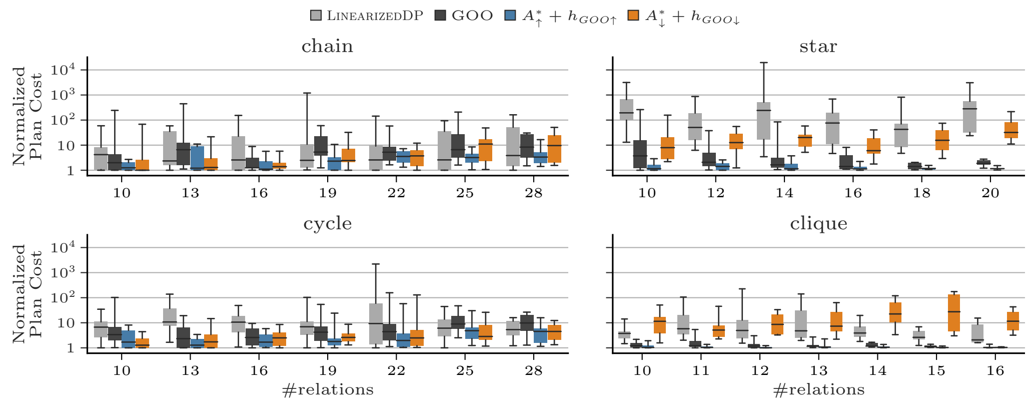
**Visualization —** In line charts, the lines connect the arithmetic means and are highlighted by their 95% confidence interval. In box plots, the boxes show the interquartile range (25% - 75%) with



(a) Optimization time of optimal algorithms.



(b) Optimization time of suboptimal algorithms.



(c) Plan cost of suboptimal algorithms, normalized to the optimal plan cost as computed by optimal algorithms, e.g., DP$_{CCP}$ .

Fig. 6. Comparison of heuristic search to state of the art.

a horizontal bar at the median (50%) and whiskers range from min to max – hence there are no outliers.

## 7.2 Comparison to State of the Art

We compare our join order optimization via heuristic search to state-of-the-art algorithms. We distinguish between optimal and potentially suboptimal algorithms. We first compare by optimization time and then, for the potentially suboptimal algorithms, we compare the computed plans by their normalized cost.

We compare the optimization times of optimal join ordering algorithms in Figure 6a. We make 4 key observations: (1) For chain and cycle, the fastest heuristic search is $A^*_\downarrow + h_{sum}$. (2) For star and clique, the fastest heuristic search is $A^*_\uparrow + h_{zero}$. (3) $A^*_\downarrow + h_{sum}$ always outperforms $A^*_\downarrow + h_{zero}$, emphasizing the importance of an informative heuristic. (4) Both DP$_{CCP}$ and TD$_{MinCutAGaT}$ are unmatched by our heuristic search on the chain and cycle topologies. On the star and clique topologies, however, our $A^*_\uparrow + h_{zero}$ performs best, at roughly $10x$ faster than $\mathrm{DP}_{\mathrm{CCP}}$ or $\mathrm{TD}_{\mathrm{MinCutAGaT}}$.

We compare the optimization times of suboptimal algorithms in Figure 6b. For all four topologies, we make the same observation: GOO is fastest, heuristic search is slowest, and LinearizedDP lies in between. On chain and cycle, both heuristic searches are equally fast, whereas on star and clique, $A^*_\uparrow + h_{GOO\uparrow}$ is significantly faster than $A^*_\downarrow + h_{GOO\downarrow}$. In addition to the optimization times, we evaluate in Figure 6c the cost of the plans computed by suboptimal algorithms, normalized to the cost of an optimal plan. We can see that $A^*_\uparrow + h_{GOO\uparrow}$ generally produces the best plans. In particular, $A^*_\uparrow + h_{GOO\uparrow}$ improves over GOO in almost all cases. Surprisingly, $A^*_\downarrow + h_{GOO\downarrow}$ produces significantly worse plans than $A^*_\uparrow + h_{GOO\uparrow}$ on star and clique topologies. LinearizedDP produces exceptionally costly plans on the star and clique topologies. This is due to LinearizedDP's greedy linearization step, that is based on IK/KBZ, a greedy algorithm to compute *optimal linear* plans [16, 18, 25]. The problem with IK/KBZ is that it assumes uncorrelated join selectivities, an artificial constraint that is not provided by our data generation (cf. Algorithm 7). Therefore, IK/KBZ computes a suboptimal linearization, which rules out many good plans for LinearizedDP.

A general observation that we can make for heuristic search is that both optimization time and plan cost are correlated to the effective branching factor $b^*$ of Figure 5: a smaller $b^*$ leads to less optimization time and a better plan. This general rule does not apply, however, when comparing two searches of opposite direction, e.g. on star topology, $A^*_\downarrow + h_{GOO\downarrow}$ achieves smaller $b^*$ than $A^*_\uparrow + h_{zero}$, but the latter is always faster. This supports our hypothesis from Section 5.3, that different types of queries are better solved by bottom-up or top-down search.

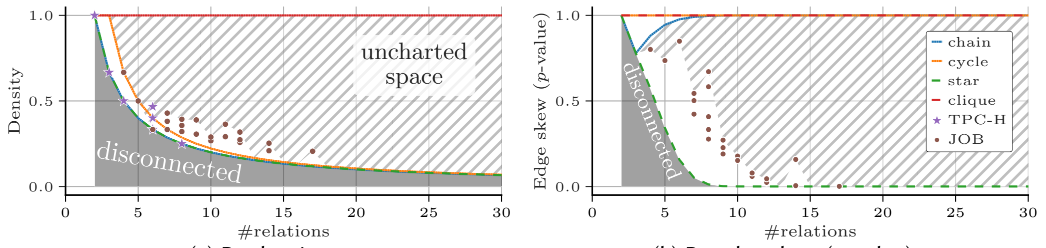
## 7.3 QGraEL: A New Benchmark for JOOP

**Motivation.** We analyze the four topologies studied in Section 7.2 together with the queries of the TPC-H and JOB benchmarks. For our analysis we introduce two measures on the query graph: *density* and *edge skew*. Density is simply defined as the graph density $D(G_Q) := \frac{2|J|}{|R|(|R|-1)}$ with $G_Q := (R, J)$, and it captures the ratio between actual edges and maximally possible edges in $G_Q$. We define edge skew as a measure for the distribution of degrees in $G_Q$, where the degree of a vertex is simply the number of edges at this vertex. We calculate edge skew as the $p$-value of the $\chi^2$ test of the actual distribution of degrees in $G_Q$ and expecting a uniform distribution of degrees. For example, a cycle has a uniform distribution of degrees, i.e. every relation has a degree of 2. The $\chi^2$ test will then compute $p = 1$ for *no edge skew*. For star, one relation has high degree while all other relations have degree 1 and $p$ will be close to 0, signaling *high edge skew*.

With measures density and edge skew, we draw the entire landscape of queries in Figure 7. For density in Figure 7a, clique is at the upper limit with a density of 1, i.e. every possible join exists in $G_Q$, and chain and star are at the lower limit, with exactly $n - 1$ joins for $n$ relations; graphs with fewer joins are disconnected. For edge skew in Figure 7b, clique and cycle have a uniform distribution of degrees and are at the upper limit of 1. With increasing number of relations, the edge skew of star increases and $p$ converges towards 0.

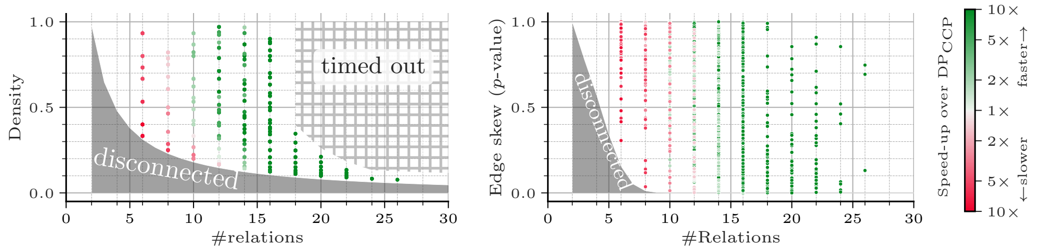
We additionally draw the queries of TPC-H and JOB into the landscape in Figure 7. We observe that all those queries have close to minimal density and high edge skew, leaving large uncharted spaces in both dimensions.

**A New Benchmark.** With this work we propose the new benchmark Query Graph Exploration Landscape (QGraEL). It systematically explores query graphs in three dimensions: number of relations, density, and edge skew. We evaluate every query in QGraEL with both $\mathrm{DP}_{\mathrm{CCP}}$ and $A^*_\uparrow + h_{zero}$ and compare their optimization times. This enables us to evaluate for which graph properties which algorithm performs better.



(a) By density. (b) By edge skew ($p$-value).

Fig. 7. Landscape of possible query graphs.



(a) By density. (b) By edge skew ($p$-value).

Fig. 8. Speed-up of $A^*_\uparrow + h_{zero}$ over $DP_{CCP}$ in QGraEL. The color encodes the relative improvement of optimization time, and it is capped at 10 in both directions. Note, that in the worst case, $A^*_\uparrow + h_{zero}$ was less than 10x slower than $DP_{CCP}$ while in the best case we achieve speed-ups >1000x.

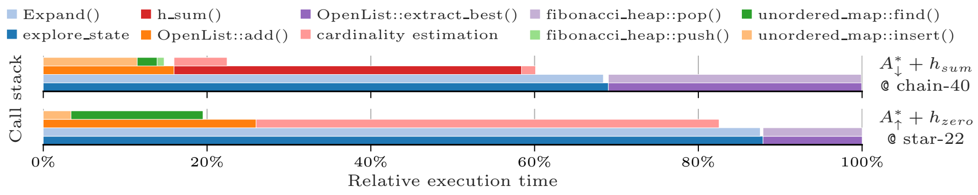


Fig. 9. Detailed running time analysis of heuristic search.

**Results.** Figure 8 shows our results, depicted along the three dimensions number of relations, density, and edge skew. We explore the landscape as much as possible, i.e. until either algorithm reaches a fixed timeout. The color encodes the improvement or deterioration of heuristic search over $DP_{CCP}$. We observe that large spaces of the landscape that have been unexplored so far are clearly dominated by heuristic search, with exceptional speed-ups of up to 1000x.

## 7.4 Detailed Evaluation of Heuristic Search

We perform an in-depth evaluation of heuristic search to understand how much the different operations contribute to the overall optimization time. We measure how much time is spent in each function via statistical profiling with the Linux PERF tool. We profile two runs: $A^*_\downarrow + h_{sum}$ on a chain query of 40 relations (`chain-40`) and $A^*_\uparrow + h_{zero}$ on a star query of 22 relations (`star-22`). We visualize the collected profiling data as a *flame graph*, that is a stacked horizontal bar chart, where one bar corresponds to one function and the width of the bar corresponds to the time spent within this function. When one function is called from another function, their bars are vertically stacked from bottom to top and in the order of the call stack. Figure 9 presents our findings. For search with $A^*_\uparrow + h_{zero}$, the heuristic is optimized out during compilation and hence does not appear in the flame graph. We can see that search spends a large share of its optimization time in EXPAND(), however

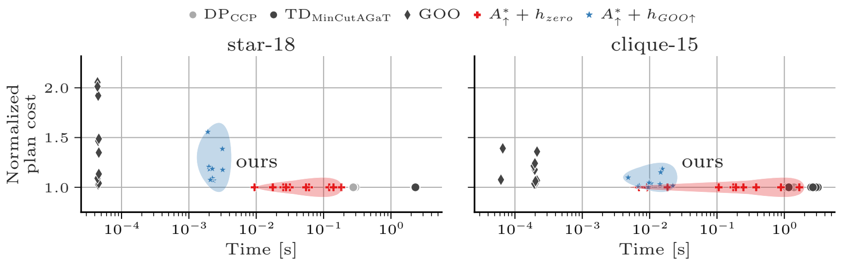


Fig. 10. Pareto frontier of optimization time vs. plan cost.

only a fraction of time is spent inside the function itself. This observation supports our claim in Section 4.2, that vertex expansion makes up for only a small fraction of overall search time. On `chain-40`, most time is spent on evaluating the heuristic and on extracting the top element from the heap. On `star-22`, more than half of the time is spent on cardinality estimation. Note, that the optimization times are relative: on `star-22` search does not spend more time on cardinality estimation than on `chain-40` but instead search on `star-22` spends *less* time on managing the open list. This is due to search on `star-22` being more goal-oriented than search on `chain-40`, hence generating less duplicate vertices. The generation of duplicates on `chain-40` leads to frequent recalculation of heuristic values without progressing further towards the goal.

## 8 CONCLUSION

With this work, we provide a sound and generic framework for join order optimization via heuristic search. Our optimizations make heuristic search practical for application in a real DBMS, as our evaluation confirms. Figure 10 shows that we are able to extend the Pareto frontier of optimization time vs. plan cost. Our optimal solution $A^*_\uparrow + h_{zero}$ outperforms state of the art by up to 2 orders of magnitude on star and clique topologies. Our suboptimal solution $A^*_\uparrow + h_{GOO\uparrow}$ provides a middle ground between GOO, which is fast but shows high variance in plan quality, and our optimal solution.

While this paper aims to be self-contained, there are many aspects or variations to our approach that did not fit into a single paper. We would like to give a glimpse of what future research may focus on: • designing more informative heuristics, potentially tuned for different query topologies, • applying different search strategies, e.g. *beam search*, *iterative deepening* $A^*$, *fringe search*, • *anytime search*, where search proceeds until a resource is exhausted or search is stopped and then retrieving the best plan found so far, or • *bidirectional heuristic search*, where search is simultaneously performed bottom-up and top-down and when both searches meet, a plan is found. We believe that our work serves as a foundation for and enables future research in the direction of computing join orders with heuristic search.

We would like to thank Karl Bringmann (MPII) for clarifying Section 2.4 & Section 3.2 and proofreading Theorem 1, Daniel Gnad (Linköping University) for supporting this work with his expertise in AI planning, and Felix Brinkmann for conducting an initial investigation on the applicability of AI planning to JOOP as part of his Bachelor's thesis.

## REFERENCES

[1] Brian Babcock and Surajit Chaudhuri. 2005. Towards a robust query optimizer: a principled and practical approach. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 119–130.
[2] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. 1995. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*. IEEE, 190–200.
[3] Sophie Cluet and Guido Moerkotte. 1995. On the complexity of generating optimal left-deep processing trees with cross products. In *International Conference on Database Theory*. Springer, 54–67.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2016. *Introduction to Algorithms.* The MIT Press.

[5] Rina Dechter and Judea Pearl. 1985. Generalized best-first search strategies and the optimality of A. *Journal of the ACM (JACM)* 32, 3 (1985), 505–536.

[6] David DeHaan and Frank Wm Tompa. 2007. Optimal top-down join enumeration. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data.* 785–796.

[7] Edsger W Dijkstra et al. 1959. A note on two problems in connexion with graphs. *Numerische mathematik* 1, 1 (1959), 269–271.

[8] Stefan Edelkamp and Stefan Schrodl. 2011. *Heuristic search: theory and applications.* Elsevier.

[9] Leonidas Fegaras. 1998. A new heuristic for optimizing large queries. In *International Conference on Database and Expert Systems Applications.* Springer, 726–735.

[10] Pit Fender. 2014. Algorithms for Efficient Top-Down Join Enumeration. (2014).

[11] Pit Fender and Guido Moerkotte. 2011. A new, highly efficient, and easy to implement top-down join enumeration algorithm. In *2011 IEEE 27th International Conference on Data Engineering.* IEEE, 864–875.

[12] Pit Fender and Guido Moerkotte. 2011. Reassessing top-down join enumeration. *IEEE Transactions on Knowledge and Data Engineering* 24, 10 (2011), 1803–1818.

[13] Frank Gray. 1953. Pulse code communication. US Patent 2,632,058.

[14] Immanuel Haffner, Marcel Maltry, Joris Nix, Jens Dittrich, and Luca Gretscher. 2023. mu*t*able. https://mutable.uni-saarland.de

[15] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107.

[16] Toshihide Ibaraki and Tiko Kameda. 1984. On the optimal nesting order for computing n-relational joins. *ACM Transactions on Database Systems (TODS)* 9, 3 (1984), 482–502.

[17] Navin Kabra and David J DeWitt. 1998. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data.* 106–117.

[18] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. 1986. Optimization of Nonrecursive Queries.. In *VLDB*, Vol. 86. 128–137.

[19] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[20] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making learned query optimization practical. In *Proceedings of the 2021 International Conference on Management of Data.* 1275–1288.

[21] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul23. 2019. Neo: A Learned Query Optimizer. *Proceedings of the VLDB Endowment* 12, 11 (2019).

[22] Guido Moerkotte and Thomas Neumann. 2006. Analysis of two existing and one new dynamic programming algorithm for the generation of optimal bushy join trees without cross products. In *Proceedings of the 32nd international conference on Very large data bases.* Citeseer, 930–941.

[23] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc Friedman, and Alekh Jindal. 2021. Steering query optimizers: A practical take on big data workloads. In *Proceedings of the 2021 International Conference on Management of Data.* 2557–2569.

[24] Thomas Neumann. 2009. Query simplification: graceful degradation for join-order optimization. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data.* 403–414.

[25] Thomas Neumann and Bernhard Radke. 2018. Adaptive optimization of very large join queries. In *Proceedings of the 2018 International Conference on Management of Data.* 677–692.

[26] Kenneth W Ng, Zhenghao Wang, Richard R Muntz, and Silvia Nittel. 1999. Dynamic query re-optimization. In *Proceedings. Eleventh International Conference on Scientific and Statistical Database Management.* IEEE, 264–273.

[27] Kiyoshi Ono and Guy M. Lohman. 1990. Measuring the Complexity of Join Enumeration in Query Optimization.. In *VLDB*, Vol. 97. 314–325.

[28] Matthew Perron, Zeyuan Shang, Tim Kraska, and Michael Stonebraker. 2019. How I learned to stop worrying and love re-optimization. In *2019 IEEE 35th International Conference on Data Engineering (ICDE).* IEEE, 1758–1761.

[29] Stuart Russell and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach* (4 ed.). Prentice Hall.

[30] Saïd Salhi. 2017. *Heuristic search: The emerging science of problem solving.* Springer.

[31] Alexander Schrijver. 2004. Combinatorial optimization: Polyhedra and efficiency (algorithms and combinatorics). *Journal-Operational Research Society* 55, 9 (2004), 1018–1018.

[32] P. Griffiths Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1989. Access Path Selection in a Relational Database Management System. In *Readings in Artificial Intelligence and Databases.* Elsevier, 511–522.

[33] Timos K Sellis. 1988. Multiple-query optimization. *ACM Transactions on Database Systems (TODS)* 13, 1 (1988), 23–52.
[34] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. 1997. Heuristic and randomized optimization for the join ordering problem. *The VLDB Journal* 6, 3 (1997), 191–208.
[35] Bennet Vance. 1998. *Join-order optimization with Cartesian products.* Oregon Graduate Institute of Science and Technology.
[36] Bennet Vance and David Maier. 1996. Rapid bushy join-order optimization with cartesian products. *ACM SIGMOD Record* 25, 2 (1996), 35–46.
[37] Florian Waas and Arjan Pellenkoft. 2000. Join order selection (good enough is easy). In *British National Conference on Databases.* Springer, 51–67.
[38] Wentao Wu, Jeffrey F Naughton, and Harneet Singh. 2016. Sampling-based query re-optimization. In *Proceedings of the 2016 International Conference on Management of Data.* 1721–1736.

Received July 2022; revised October 2022; accepted November 2022