Monotone Branch-and-Bound Search for Restricted Combinatorial Auctions

Citation	Lai, John, and David Parkes. 2012. Monotone branch-and-bound search for restricted combinatorial auctions. Proceedings of the 13th ACM Conference on Electronic Commerce (EC '12), June 4-8, 2012, Valencia, Spain, 705-722. New York, NY: ACM Press.
Published Version	doi:10.1145/2229012.2229067
Accessed	February 19, 2015 1:11:16 PM EST
Citable Link	http://nrs.harvard.edu/urn-3:HUL.InstRepos:11956915
Terms of Use	This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Open Access Policy Articles, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#OAP

Monotone Branch-and-Bound Search for Restricted Combinatorial Auctions

JOHN K. LAI, Harvard University School of Engineering and Applied Sciences
DAVID C. PARKES, Harvard University School of Engineering and Applied Sciences

Faced with an intractable optimization problem, a common approach to computational mechanism design seeks a polynomial time approximation algorithm with an approximation guarantee. Rather than adopt this worst-case viewpoint, we introduce a new paradigm that seeks to obtain good performance on typical instances through a modification to the branch-and-bound search paradigm. Incentive compatibility in single-dimensional domains requires that an outcome improves monotonically for an agent as the agent's reported value increases. We obtain a monotone search algorithm by coupling an explicit sensitivity analysis on the decisions made during search with a correction to the outcome to ensure monotonicity. Extensive computational experiments on single-minded combinatorial auctions show better welfare performance than that available from existing approximation algorithms.

Categories and Subject Descriptors: G.1.6 [**Mathematics of Computing**]: Optimization—*Integer Programming*; J.4 [**Computer Applications**]: Social and Behavioral Sciences—*Economics*

General Terms: Algorithms, Economics

Additional Key Words and Phrases: Computational Mechanism Design, Branch and Bound, Combinatorial Auctions

1. INTRODUCTION

Given a system of self-interested agents, each with private information about their preferences, and a set of outcomes, the problem of mechanism design is to select an outcome with desirable properties despite the ability of agents to misreport their preferences. Computational mechanism design (CMD) also insists on computational efficiency, which is a significant concern in domains such as combinatorial auctions (CAs), where the winner determination problem is NP-hard. Although typical instances of NP-hard problems such as combinatorial auctions can be routinely solved through the use of heuristic search such as branch-and-bound search [Sandholm et al. 2005; Andersson et al. 2000], a common theme in CMD is to insist on worst-case polynomial time algorithms, and look for algorithms for which there is theoretical support through worst-case approximation guarantees.

In the context of *single-minded* CAs, where each agent is interested in exactly one bundle, Lehmann et al. [2002] provide a greedy algorithm and associated payment rule with a $\sqrt{m}$ welfare guarantee (relative to the optimal welfare), where m is the number of items being allocated, and a matching lower-bound. More recently, Mu'alem and Nisan [2008] provide an approximation for the special case of *known* single-minded CAs with guarantee $\epsilon\sqrt{m}$ for any fixed $\epsilon > 0$, with runtime that is exponential in $1/\epsilon^2$. In a known single-minded CA, the bundle is known to the mechanism, transform-

EC'12, June 4–8, 2012, Valencia, Spain. Copyright 2012 ACM 978-1-4503-1415-2/12/06...$10.00.

ing this into a *single-dimensional* mechanism design problem.[1] However, if incentives were not a concern, we have more sophisticated algorithms such as Branch-and-Bound (BnB) search, which can efficiently find optimal solutions to the winner determination problem on typical instances. Following a research agenda on *heuristic mechanism design* [Parkes 2009], we seek to leverage heuristic algorithms such as BnB search for the purpose of CMD.

BnB search is a canonical method for solving optimization problems that are formulated as integer programs (IPs). Search proceeds by branching on decisions in regard to whether or not an agent is allocated ("branch"), and looking to prune large parts of the search space through linear program (LP) relaxations ("bound"). In cases where it is too computationally expensive to compute the optimal solution, an *optimality tolerance* $\gamma \in (0, 1]$ is adopted, and search is terminated when a solution is identified that is proven to be within multiplicative fraction γ of the optimal solution. We will typically consider BnB with $\gamma < 1$ in the present paper.

But therein lies the core problem in combining BnB with mechanism design:

(i) the canonical Vickrey-Clarke-Groves mechanism need not be truthful when coupled with an approximate solution to a welfare optimization problem, and

(ii) the allocation generated by BnB search with $\gamma < 1$ need not be *monotone*, in the sense that an agent might go from winning at some bid value w_i to losing at some bid $w'_i > w_i$.

BnB search is monotone for $\gamma = 1$ because it computes the optimal allocation. But monotonicity can fail with BnB when $\gamma < 1$, because a higher agent value can trigger a different search decision somewhere in the search tree, eventually leading to the search terminating with an alternate solution that is within a factor γ of optimal but does not include the agent.

Correcting this failure of monotonicity, we follow an approach introduced by Parkes and Duong [2007] in a different context. Given an instance, we check to see whether agent i allocated at bid w_i becomes unallocated for any bid $w'_i > w_i$ (fixing the other bids.) If this occurs, then the outcome is "corrected" (or *ironed*) such that the agent is not allocated at bid w_i. By doing this for all inputs, we achieve *monotone BnB search* (and thus incentive compatibility). Moreover, the approach retains good welfare if the original search algorithm is monotone for most agents on most inputs.

The technical challenge is to find an efficient method to trace the effect on the outcome of BnB search as the bid value of an agent is increased, taking each agent in turn. From the perspective of an IP, we are increasing an objective coefficient and tracing the effect on decisions made during BnB search (e.g., branch decisions and pruning decisions.) The technical innovations involved in making this sensitivity analysis of BnB search efficient include:

- An efficient technique to identify the next highest objective value coefficient at which a different search decision would be made for a given BnB search state.
- Caching search states to avoid re-running early steps of BnB search that remain the same when testing higher objective value coefficients.
- Leveraging structure of BnB search to identify sufficient conditions that ensure that agent i is allocated in any BnB solution, and terminating sensitivity checks early when this is the case.

[1] While the known single-minded assumption is restrictive, Lehmann et al. [2002] describe a pollution rights auction where companies are bidding for the right to emit certain chemicals into the air, and the pollution profiles of the companies are known. They also describe communication network settings where bidders own nodes in the network and wish to connect their nodes. If there is only a single path available between any pair of nodes, then bidders are single-minded. If it is also public knowledge which companies own which pairs of nodes, then this becomes a known single-minded setting.

- Caching of LP solutions to avoid expensive re-computations when the solutions would not have changed.
- Making BnB search more monotone by adopting a bucketing approach to fractional variables in deciding which variable to branch on, and through a discrete transformation on the inputs.

We implement our technique and report experimental results based on the well-studied "legacy" distributions.[2] In particular, we focus on the L4 (decay) distribution, which has been shown in the literature to generate hard winner determination problems [Leyton-Brown et al. 2000; Sandholm 2002; Sandholm et al. 2005]. We find sets of randomly generated instances from the L4 distribution where the best parameterizations of our monotone BnB algorithm yield better welfare than the approximation mechanisms of Lehmann et al. [2002] and Mu'alem and Nisan [2008].

Additionally, the best parameterizations of monotone BnB (and for an optimality tolerance $\gamma < 1$ at which welfare is better than existing approximation mechanisms) have better runtime than optimal BnB. Monotone BnB is also fully parallelizable in the number of allocated agents while the same is not true of optimal BnB. The fully parallelized runtime cost of monotone BnB is significantly smaller than that of optimal BnB for the best parameterizations of monotone BnB and instances we consider. Though our experimental results depend crucially on these input distributions, we believe they demonstrate the potential of the general approach and the specific application to BnB search.

In addition, while earlier work has developed techniques for the sensitivity analysis of optimal solutions to IPs [Marsten and Morin 1977; Feautrier 1988], we are not aware of earlier work on the sensitivity of BnB search when used with an optimality tolerance. For this reason, we also provide some analysis of the kinds of decisions that tend to change during search and the kinds of monotonicity failures that we see on our instances. We see that on our test instances, the most common changes result from a pivot to a new LP solution, which causes a change in the branch variable selected. Due to space constraints, we omit some detailed descriptions and results from this version of the paper and point the interested reader to the full version.

1.1. Related Work

We follow earlier work of Parkes and Duong [2007] and Constantin and Parkes [2009], who have applied so-called "computational ironing" to online stochastic combinatorial optimization (OSCO). BnB search is more complex algorithmically than the OSCO algorithms studied in this earlier work, and requires new technical contributions in finding an efficient coupling with the approach of computational ironing.

Also thematically related to heuristic mechanism design is the GROWRANGE method of Parkes and Schoenebeck [2004], which provides an anytime algorithm for welfare optimization in general CAs by expanding the range of a VCG-based algorithm, while allowing for a time-based interruption by the center (although without providing full strategyproofness.)

Some other papers provide techniques for converting non-truthful approximation algorithms into Bayes-Nash incentive compatible (BIC) mechanisms with essentially the same approximation factor. Lavi and Swamy [2005] give such a construction for problems where the optimization problem can be written as an integer program, but usefully solved as a linear programming (LP) relaxation. This is the key difference from our work: we do not rely on the existence of a good approximation gap for a LP relax-

[2] We did not use the named CATS distributions (matching, paths, regions, scheduling) as there is no straightforward way to adapt these to the single-minded setting.

ation (moreover, we work with dominant-strategy IC rather than BIC.) On the other hand, Lavi and Swamy handle a problem of (welfare maximizing) *multi-dimensional* mechanism design. Their construction yields a BIC mechanism with approximation guarantee $O(\sqrt{m})$ when applied to (general) CAs. Hartline and Lucier [2010] and Hartline et al. [2011] provide a general approach, for both single-dimensional and multi-dimensional domains, for converting any approximation algorithm into a BIC mechanism with the same approximation guarantee on welfare. This differs from our work as we target dominant-strategy IC. Additionally, we are not aware of any computational studies of these suggested approaches.

2. COMBINATORIAL AUCTIONS

In the combinatorial auction (CA) problem, there is a set N of agents and set G of items, with $|N| = n$, $|G| = m$. Each agent has a private *valuation function* $v_i : 2^G \to \mathbb{R}_{\geq 0}$ which expresses the agent's value for each possible bundle of items. A *valuation profile* $(v_1, \ldots, v_n) = v$ consists of a valuation function for each agent. It will be useful to write a valuation profile from the perspective of agent i as $v = (v_i, v_{-i})$, where v_i gives agent i's valuation function and v_{-i} refers to the valuation function of all other agents.

An *allocation* a is an assignment of items to each agent, and a *feasible allocation* requires that no item is given to more than one agent. An *allocation function* g maps from a reported valuation profile to an allocation, and a *payment function* p maps from a reported valuation profile to a payment for each agent. By g_i we denote the bundle assigned to agent i and p_i denotes the payment of agent i. A *mechanism* is a pair (g, p).

Agents have *quasi-linear* utilities; i.e., if agent i receives bundle $T \subseteq G$ and pays q, then the agent's utility is $v_i(T) - q$. The utility of agent i under mechanism (g, p) and reported valuations $v' = (v'_i, v'_{-i})$ is $u_i(v') = v_i(g_i(v')) - p_i(v')$, where v_i is the agent's true valuation function.

Because agent valuations are private, we are interested in mechanisms for which it is weakly beneficial for agents to truthfully report their valuation functions. A mechanism (g, p) is *truthful* if for every agent i, for every valuation function v_i and alternate report v'_i, for all valuations v_{-i} of other agents, $u_i(v_i, v_{-i}) \geq u_i(v'_i, v_{-i})$.

For many settings such as spectrum auctions or auctions of landing slots at airports, we would like the allocation to maximize the *social welfare* or total value to the bidders. The *social welfare* of an allocation a with respect to values $(v_1, \ldots, v_n)$ is $W(a, v) = \sum_{i=1}^{n} v_i(a_i)$. A mechanism (g, p) is *efficient* if for all v, $g(v) \in \arg\max_{a:a\ isfeasible} W(a, v)$. We also require that our mechanisms never give agents negative utility. A mechanism (g, p) is *individually rational* if for all v, for all i, $v_i(g_i(v)) - p_i(v) \geq 0$.

2.1. Known single-minded CA

In known single-minded CAs each agent has a target bundle T_i, known to the mechanism, and a value $w_i > 0$ for this bundle. We thus refer to agent reports as being this single value w_i, rather than a report of an agent's entire valuation function v_i. Also, we consider deterministic allocation functions, so we can assume that $g_i(w_i, w_{-i}) \in \{0, 1\}$ corresponding to whether or not the agent is allocated its target bundle.

Definition 2.1. An allocation function g is monotone in a known single minded-domain if $g_i(w_i, w_{-i}) = 1 \Rightarrow g_i(w'_i, w_{-i}) = 1$ for all $w'_i \geq w_i$.

For deterministic allocation functions, we have the following well-known observation:

THEOREM 2.2. *[Myerson 1981] Given allocation function g, and a known single-minded domain, there exists a payment function p that makes (g, p) truthful iff g is monotone.*

In fact, once the allocation function is known, the payment function can be computed by finding the "critical value" at which an agent starts receiving its target bundle. As a result, for single-dimensional settings such as known single-minded CAs, the problem of constructing truthful mechanisms can be reduced to that of finding monotone allocation functions.[3]

Known single-minded CAs are a special case of the more general class of downward closed environments.

Definition 2.3. A single-dimensional mechanism design environment is *downward closed* if a feasible allocation is exactly described by a set of agents allocated, and any subset of a feasible allocation is feasible.

In the known single-minded setting, the feasible sets are the sets of agents whose target bundles share no intersection and the downward closed property holds because not all items need to be allocated. While we focus on known single-minded CAs, the general ironing procedure developed in Section 3 applies to single-dimensional, downward closed environments.

3. IRONING, DISCRETIZATION AND A FIRST APPROACH

We first describe the very basic approach to making heuristic algorithms monotone for downward closed domains. In the next sections, we propose techniques to reduce the computational overhead in the particular context of BnB search.

The basic idea of ironing is straightforward. We first compute the set of allocated agents using our allocation algorithm at the current values. We then perform sensitivity analysis on the set of allocated agents. For each allocated agent, we check if the agent would still be allocated under the allocation algorithm for all higher reported values. If an agent becomes deallocated for higher reported values, then we must deallocate the agent since this indicates a non-monotonicity in the provided allocation function. This general procedure is described as *ironed-alloc* in Figure 1. We focus on the allocation function in the body of the paper, but the same ideas can be applied to compute payments for allocated agents by performing downward sensitivity rather than upward sensitivity.

THEOREM 3.1. *The* ironed-alloc *procedure is monotone and feasible for downward closed domains.*

PROOF. An agent is allocated in *ironed-alloc* only if the agent is allocated at its current value and all higher reported values. If this is the case, then *ironed-alloc* would still have allocated the agent for higher reports. □

If the underlying allocation function is monotone everywhere, then *ironed-alloc* will be the same as the underlying algorithm. If it is not, then *ironed-alloc* may sacrifice welfare (since it must deallocate some agents) in order to preserve monotonicity.

As stated, *ironed-alloc* applies to continuous type domains as long as we have a method for *sensitivity checking*; i.e., to determine whether an agent will become deallocated for any higher reports. However, such a procedure may not always be available, and even when it is, implementing such procedures in practice may introduce an implicit discretization.[4]

[3] In the context of the framework of the current paper, once an allocation has been confirmed for an agent by performing a check of monotonicity for all higher reports the agent could have made, then a parallel, *downward* sensitivity check is performed to find the *first* smaller value at which the agent would no longer be allocated.

[4] Initially, we developed our sensitivity checking procedure for continuous values. We identified sensitivity points w_1 based on whether the search might change for any value strictly greater than w_1 (open) or for any

```
ironed-alloc( alloc-func , values)
    allocated = alloc-func (values)
    for agent ∈ allocated
    do
        if is-deallocated-at-higher-values(agent)
            then allocated ← allocated \ agent

    return allocated
```

```
discretized-ironed-alloc( alloc-func , values, β)
    for value ∈ values
    do value = ⌊value/β⌋β
    allocated = alloc-func (values)
    for agent ∈ allocated
    do
        if is-deallocated-at-higher-values(agent)
            then allocated ← allocated \ agent

    return allocated
```

Fig. 1. General procedures for making an allocation function monotone.

For this reason, we introduce a discretized version of *ironed-alloc* that is monotone in the original domain, even if the original domain is continuous, and still results in payments that are individually rational. The procedure mimics *ironed-alloc*, except that agent bids are rounded down to the nearest grid size β (Figure 1).

THEOREM 3.2. *Procedure* discretized-ironed-alloc *is monotone and admits individually rational payments.*

PROOF. The proof of monotonicity is the same as Theorem 3.1. To see that payments are individually rational, recall that given a monotone allocation function, the payment of an agent is the lowest value at which the agent would still be allocated. Suppose that an agent is allocated when bidding w_i. This means that the agent would also be allocated with bid $\lfloor w_i/\beta \rfloor \beta \leq w_i$, so the agent's payment is at most w_i. □

With grid size β, we can obtain a procedure *is-deallocated-at-higher-values* by testing all multiples of β that are greater than the agent's current value, to see whether the agent would still be allocated. Because *discretized-ironed-alloc* rounds values down to multiples of β prior to sending them to the allocation function, this will capture all possible points where the agent could have become deallocated. We refer to this as the *brute force* sensitivity method.

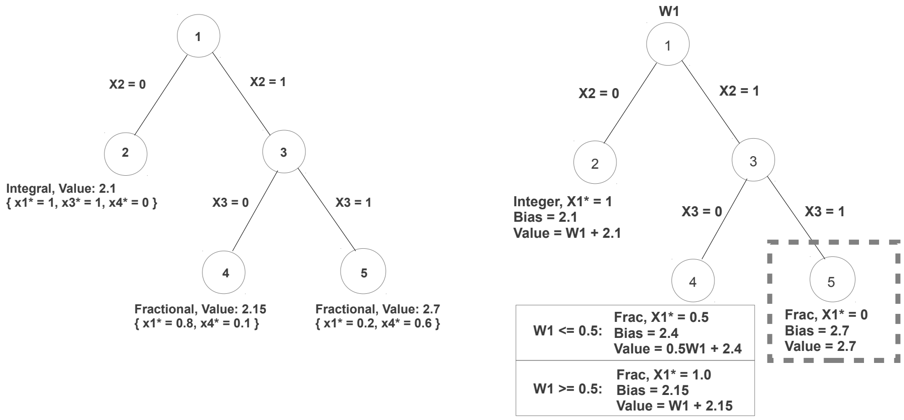
There is an interesting trade-off in using discretization in the context of ironing. On one hand, the allocation function no longer accesses exact agent values, which can result in allocations with lower welfare compared to the allocations computed using the true values. On the other hand, adopting a discretization may actually improve the "ironed" welfare because there are fewer points where the algorithm is required to still allocate the agent, and as a result, the underlying algorithm may become more monotonic and deallocate fewer agents.

4. BRANCH-AND-BOUND SEARCH FOR COMBINATORIAL AUCTIONS

An empirically effective way to find an allocation with good welfare for CAs is to formulate the problem as an integer program (IP) and use BnB search. We describe the essentials of this approach in this section.

In the known single-minded CA setting, where each agent is interested in a single bundle T_i and reports value w_i, we can write the following winner determination IP

value weakly greater than w_1 (closed). To handle open points, we needed to introduce a parameter ϵ to jump the agent's value to $w_1 + \epsilon$ when running counter-factuals. This discussion will be clearer after Section 5.



(a) (b)

Fig. 2. (a) A simple illustration of a branch-and-bound search tree. (b) An illustration of the augmented search state and *get-sens-single-state*.

(WDIP) to solve for the welfare-maximizing allocation:

$$\text{maximize} \sum_{i=1}^{n} w_i x_i \tag{1}$$

$$\text{subject to} \quad \sum_{i:j \in T_i} x_i \leq 1, \; 1 \leq j \leq m \tag{2}$$

$$x_i \in \{0, 1\}, \; 1 \leq i \leq n \tag{3}$$

The linear programming (LP) relaxation of this IP is the same program, except with the integer constraints (3) replaced by inequalities of the form $0 \leq x_i \leq 1$. Given this, branch-and-bound (BnB) is a *tree search* technique that uses the relationship between an IP and its LP relaxation to prune parts of the search tree. We will focus on the case where the variables are binary (0 or 1) since the IPs we consider will have this form.

The basic components of the search are the nodes in the search tree. Each node k stores an integer partial assignment, i.e. $t = \{x_2 = 0, x_4 = 1\}$, along with a solution $\{x_1^*, \ldots, x_n^*\}$ to LP_t, where LP_t is the LP relaxation of WDIP, with extra constraints added to enforce t. Let $t(k)$ denote the partial assignment stored in k. With a slight abuse of notation, we say $j \in t(k)$ if x_j is set to 0 or 1 in $t(k)$.

Let the *value of an LP* be $\sum_{i=1}^{n} w_i x_i^*$, where $x_1^*, \ldots, x_n^*$ is the solution to the LP, and the *value of a node* $k = val(k)$ be the value of its LP relaxation. Because the value of an LP relaxation is an upper bound on its associated IP, $val(k)$ is an upper bound on any integer solution that agrees with $t(k)$. A solution $\{x_i^*\}$ is *integral* if $x_i^* \in \{0, 1\} \; \forall i$, and *fractional* otherwise. In Figure 2(a), node 1 corresponds to an empty partial assignment, while nodes 2 and 3 correspond to partial assignments $\{x_2 = 0\}$, $\{x_2 = 1\}$ respectively. This indicates that x_2 is set to 0 in node 2 and all its children, while x_2 is set to 1 in node 3 and all its children.

A *search tree* has a *root node* with an empty partial assignment, and other nodes are either an internal node with two children or a leaf node. The left child of an internal node k corresponds to adding $x_j = 0$ to $t(k)$ while the right child corresponds to adding $x_j = 1$ to $t(k)$, for some $j \notin t(k)$. An important property of a search tree is that any integer solution agrees with the partial solution in exactly one leaf of the search tree, i.e. the leaves of any search tree partition the space of possible integer solutions.

The *search state* s is a collection of nodes, and corresponds to the leaf nodes in a valid search tree. $I(s)$ denotes the integral nodes associated with s, $F(s)$ the fractional nodes associated with s, and $K(s) = I(s) \cup F(s)$ all nodes associated with s. In Figure 2(a), the search state consists of nodes 2, 4, 5, with node 2 integral and nodes 4 and 5 fractional. Given a search state s, we define the $dec(s)$ to be the *decision* associated with s. To specify the decision, we assume that BnB is being run to an *optimality tolerance* $\gamma \in (0, 1]$, where $\gamma = 1$ represents full optimality. The search decision consists of:

(1) Whether or not to terminate the search because a solution with welfare at least γ times the optimal has been found.
(2) If the search is terminated, a node $k \in I(s)$ that has the highest value.
(3) If the search is not terminated,
    (a) A node $k \in F(s)$ to be selected.
    (b) A variable x_j to be branched.

The crux of BnB lies in how the decision associated with s is computed. We define:

$$UB(s) = \max_{k:k \in F(s)} val(k), LB(s) = \max_{k:k \in I(s)} val(k)$$

If $\gamma \cdot UB(s) \leq LB(s)$, then terminate, and select a node $k \in I(s)$ that has value $LB(s)$. If $\gamma \cdot UB(s) > LB(s)$, then select a node $k \in F(s), \gamma \cdot val(k) > LB(s)$ to be explored (*select-node*) along with a variable x_j to be branched (*branch-variable*). Altogether, the BnB procedure proceeds as:

(1) Initialize s to be a single node corresponding to the empty partial assignment.
(2) Repeat until termination:
    — Compute $dec(s)$.
    — If terminate, return the integral solution in the node given by $dec(s)$ and terminate.
    — If not terminate, update s by replacing the node given by $dec(s)$ with two children corresponding to branching the variable x_j given by $dec(s)$.

There are various choices for how to implement the *select-node* and *branch-variable* functions. For instance, *select-node* can choose the deepest node, breaking ties by value, (*depth-first*) or choose the node with the highest LP solution value (*breadth-first*) or alternate between the two. A popular choice for *branch-variable* is to select the most fractional variable in the LP solution, but other choices are also possible (see e.g. Chapter II.4 in Nemhauser and Wolsey [1998]). The best choices for these functions are typically domain specific. In our work, we choose depth-first for *select-node* until an integral node is found, after which point we use breadth-first. For *branch-variable*, we focus on variants of selecting the most fractional variable.

Upon termination, BnB will return a solution with welfare at least γ times the optimal. This is true because at each step $\max(LB(s), UB(s))$ is an upper bound on the value of any integer solution to WDIP because of the admissibility (or optimistic) estimate of value that comes from the use of LP relaxations and because the nodes in s partition the space of integer solutions.

5. OPTIMIZED SENSITIVITY CHECKING FOR BRANCH-AND-BOUND SEARCH

In this section, we demonstrate an optimized sensitivity checker (i.e. an implementation of *is-deallocated-at-higher-values*) that takes advantage of the structure of BnB search. In what follows, we assume that we are performing sensitivity checking in the context of *discrete-ironed-alloc*, and can therefore assume that input values are multi-

```
*optimized-is-deallocated-at-higher-values*( **alloc-func** , $values, agent, \beta$)
  $sens\text{-}value = values(agent)$
  **while** $sens\text{-}value < max\text{-}value$
  **do** $values(agent) = sens\text{-}value$
      $alloc, next\text{-}value = get\text{-}sensitivity($ **alloc-func** $, values, agent, \beta)$
      **if** $agent \notin alloc$
         **then return** $true$
      $sens\text{-}value = next\text{-}value$

  **return** $false$
```

Fig. 3. An optimized procedure for checking whether an agent becomes deallocated for higher values.

ples of β.[5] For the duration of this section, we assume without loss of generality that we are performing sensitivity checking for agent 1.

Rather than re-run the search for every higher multiple of β, we would ideally like to skip multiples of β that provably continue allocating agent 1 in the solution returned by BnB. The core of such a procedure would consist of a function *get-sensitivity* that runs BnB with agent 1's value set to w_1, but in addition to returning an allocation, returns the next value $w_1' > w_1$ for which we should re-run the search. We could then use the following procedure (summarized in Figure 3) as a replacement for the brute force sensitivity checker. We first run *get-sensitivity* with agent 1's reported value. This returns the allocation BnB would have returned, along with the next higher value w_1' at which the allocation might change. We set agent 1's value to w_1', and re-run *get-sensitivity*. If the allocation returned continues to allocate agent 1, then continue the process. Terminate if an allocation returned does not allocate agent 1 or if the next highest value exceeds the maximum allowed value.[6]

The next two sections are devoted to defining *get-sensitivity*. We first examine how a change in w_1 affects a specific node in the search state, and we then use these observations to provide an implementation for *get-sensitivity*.

5.1. Impact of a change in value on a Search Node

We first examine how a *node* in the search state (recall, associated with an LP) changes when agent 1's reported value increases. We separate these changes into two types.

5.1.1. Solution value changes. As a agent 1's reported value increases, the solution to the LP relaxation in a given node may not change, but the value of the solution will change if $x_1^* > 0$ in the solution. If we assume that the solution does not change, then we can easily track how the solution value changes as the agent's reported value increases. Let $x_1^*, \ldots, x_n^*$ be the fractional solution to the LP relaxation at a node. The solution value as a function of agent 1's report w_1 is $val(w_1) = w_1 x_1^* + \left(\sum_{i=2}^{n} w_i x_i^*\right)$, where the expression within the parenthesis does not depend on w_1. We call x_1^* the *coefficient* and the term in parenthesis the *bias* of the node.

[5] To avoid LP degeneracy (which is problematic for sensitivity because we are unsure which solution will be picked for higher agent values), in our experiments, we add a random value in $[0, \beta)$ to the discretized agent values. This perturbation is independent of an agent's report and thus does not affect truthfulness. In practice, to maintain individual rationality, one would want to subtract a random value, but our implementation adds a random value to avoid special casing perturbations that lead to negative values. This should not have any substantive effect on our experimental results, as properties of the solution are always computed with respect to the original values prior to any discretization or perturbation.

[6] If the possible values are uncapped, we could always set a very high max value, and treat (the rare case of) any reports greater than this value as being the max value.

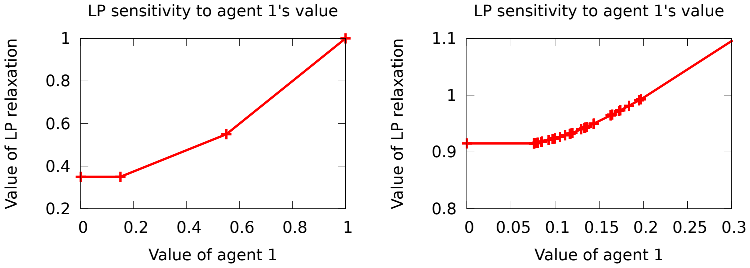


Fig. 4. Sensitivity of the LP solution value to changes in agent 1's value.

5.1.2. LP solution changes. As an agent's reported value increases, the solution to the LP relaxation at the node can change. LPs have the property that solutions lie at corners of the polyhedron formed by the constraints of the LP. The solution stays the same for a range of values, until the agent's value reaches a critical point where the LP is degenerate, and two solutions share the same value. Above this, the new solution becomes the unique optimal solution. The literature on LPs provides simple computational procedures for computing the sensitivity of a LP solution to coefficients in the objective function (see e.g. Section 5.1, [Bertsimas and Tsitsiklis 1997]).

5.1.3. Worked example. Figure 4 illustrates these two types of changes by charting how the value of the LP relaxation changes as the value of agent 1 moves from 0 to 1. The figure on the left represents the simple case where the LP corresponds to the root node of the following instance with 3 agents and 5 goods: Agent 1 desires $\{A, D, E\}$, agent 2 desires $\{A, B\}$ at 0.2, and agent 3 desires $\{B, E\}$ at 0.35. When agent 1's value is 0, the LP solution sets $x_2^* = 0, x_3^* = 1$ and has value 0.35. When agent 1's value reaches 0.15, the LP solution changes to $x_1^* = 0.5, x_2^* = 0.5, x_3^* = 0.5$. At this point, the first and second solutions have the same value. When agent 1's value reaches 0.55, agent 1 becomes fully allocated with $x_1^* = 1$. In $[0, 0.15]$, the value of the LP solution does not depend on agent 1's value. In $[0.15, 0.55]$ the value of the LP solution has slope 0.5, and in $[0.55, 0.1]$, the value of the LP solution has slope 1.0. The figure on the right depicts a more complicated example for the root node LP of an instance with 300 agents. Each marked point on the graph depicts a point where the LP solution changes, and within marked points, the value of the LP solution is linear in the value of agent 1, with the slope governed by the assignment x_1^* in the LP solution. This example is representative of how the LP solution can be quite sensitive to changes in agent 1's value. As is the case in these examples, the slope increases as agent 1's value increases.

5.2. Isolating major changes and defining *get-sensitivity*

Having discussed the two ways in which a change in an agent's value affects a *node* in a search state, we present our implementation of *get-sensitivity*. The *get-sensitivity* method runs the search with agent 1's value set to w_1, but in addition to running the search, returns a higher value $w_1' > w_1$ at which to re-run the search. The guarantee is that if *get-sensitivity* returns w_1', then setting agent 1's value to any multiple of β in (w_1, w_1') and running BnB must still result in an allocation that contains agent 1.

As we run the BnB procedure, we can ask at each search state, what higher value of agent 1's report might cause a change to occur in the search? Therefore, we reduce *get-sensitivity* to the simpler problem of figuring out the value at which the decision at a single state would change. The minimum of these sensitivity values across all search states processed in BnB provides the next value to be returned by *get-sensitivity*. We call this single search state procedure *get-sens-single-state*.

A first attempt at *get-sens-single-state* would consider any higher value at which any aspect of a search state changes (e.g., the value of solution at any associated node in the state.) But this would trigger a large number of changes since the number of nodes scales with the number of steps in the search.

Instead, we focus (for a given state s) on identifying the next higher value at which the search decision changes (i.e., *whether or not we terminate, change the identity of the integral node in the case of termination, or change the selected node or branch variable if we do not terminate*).

In what follows, we assume that *select-node* is breadth-first and chooses the fractional node with highest value, although we can adapt the procedure to other choices of *select-node* (see full version). In order to find the lowest point where the decision associated with s changes, we introduce an augmented search state. Let $w_{1\ temp}$ be agent 1's value being currently considered.[7] The augmented search state adds the following information to a search state:

(1) For each node in the state, compute the coefficient and bias for $w_{1\ temp}$.
(2) Compute the best integral node for $w_{1\ temp}$.
(3) Compute the best fractional node for $w_{1\ temp}$.

The purpose of this augmented search state is to allow us to understand how the search state changes as agent 1's value increases further from $w_{1\ temp}$, and thus when the search decision changes. As agent 1's value increases, we know that the values at the various nodes in the state will each increase linearly based on the coefficient of that node. With the assumption that *select-node* is breadth-first, the decision at s depends only on a comparison between the best integral node and the best fractional node. Figure 2(b) gives an example of the augmented search state.

The method *get-sens-single-state* for augmented state s' repeatedly finds the next lowest value $w'_1 > w_{1\ temp}$ where one of the following changes occurs:

(1) The best fractional node changes identity.
(2) The best integral node changes identity.
(3) The value of the best integral node crosses the value of the best fractional node (possibly multiplied by γ if we are not running to optimality).
(4) The LP solution of some node changes.

For each such value, the method considers the next higher value on the discrete grid, and at this value checks to see whether the search decision would actually change at this state. If it does, then this becomes the relevant sensitivity value for this state—the first value at which the search decision first changes. We defer a full description and analysis of the correctness of *get-sens-single-state* to the full version. We provide an example here.

Figure 2(b) shows the augmented search state s' for different ranges of w_1, agent 1's value. The LP solutions in nodes 2 and 5 are not dependent on w_1 while the solution in node 4 is dependent on w_1. When $w_1 \leq 0.5$, the LP solution in node 4 assigns $x^*_1 = 0.5$ with a bias of 2.4. When $w_1 \geq 0.5$, the LP solution in node 4 assigns $x^*_1 = 1.0$ with a bias of 2.15.

We now analyze *get-sens-single-state*. Suppose $w_1 = 0.25, \beta = 0.01$. Node 2 is the best integral node, while node 5 is the best fractional node. At $w'_1 = 0.5$, event 4 is triggered as the LP solution in node 4 changes to a solution with $x^*_1 = 1.0$. Note that at

[7] We make a distinction versus w_1 since $w_{1\ temp}$ can be a value higher than w_1, that we are currently examining for sensitivity purposes.

$w'_1 = w'_{1\beta} = 0.5$, at node 4, the value of the previous LP solution is equal to the value of the new LP solution ($0.5 \cdot 0.5 + 2.4 = 0.5 + 2.15$). No further updates are needed for s' as node 4's value (2.65) is still less than node 5's. Though the LP solution in node 4 has changed, the decision remains to select node 5 and continue searching. The same branch variable will be selected because the LP solution for node 5 has not changed. As a result, *get-sens-single-state* will continue, setting $w_{1\ temp}$ to 0.5. Assuming that no LP solutions change, the next event triggered will be event 1 at value $w'_1 = w'_{1\beta} = 0.55$. At this value, node 4 will overtake node 5 as the best fractional node (the value of the LP solution in node 4 reaches 2.7 while the value of the LP solution in node 5 stays at 2.7). The decision associated with s' will now change because node 4 will be selected as the next node to be explored. *get-sens-single-state* will return 0.55.

The example demonstrates the key ideas of *get-sens-single-state*. Not all events will lead to changes in the decision, but we need to capture all of these events to make sure that s' reflects the true state if agent 1 were to report these higher values. In particular, event 4 is very important because it makes sure that the coefficient and bias values are valid for the range of agent 1's values being considered.

5.3. Hot restart and Inference

5.3.1. Hot restart. With *get-sens-single-state*, we can now fully instantiate *get-sensitivity* and *optimized-is-deallocated-at-higher-values*. To check whether an agent becomes deallocated, take the minimum next value returned by calls to *get-sens-single-state* from every state in the BnB search and re-run BnB search with the agent's value updated to the minimum next value. This procedure may already outperform brute force sensitivity because we may skip over many higher multiples of β that would not have changed any search decision.

However, we can further improve performance with the following optimization. Suppose that the minimal next value w'_1 returned by all the calls to *get-sens-single-state* across all decisions made in the search occurs at step 1000 in the search. This implies that the decisions at steps 1 through 999 would not have changed if agent 1's value is updated to w'_1. As a result, we need not re-run all these steps of the search. We can save the state after step 999 and rerun the search from this point. This inspires the following modified procedure for *get-sensitivity*.

Let $w_{1\ min}$ represent the lowest next-highest value returned by any call to *get-sens-single-state* thus far in the search. Whenever a search decision is made, *get-sens-single-state* is called. If the next value returned is weakly greater than $w_{1\ min}$, then ignore it (the search decision would have changed earlier in the search). If the next value returned is less than $w_{1\ min}$, then reduce $w_{1\ min}$ to this value, and take a snapshot of the search state. Push this snapshot, along with $w_{1\ min}$, onto a list of search states from which to re-run. We refer to this as *hot restart*.

Figure 5 gives a way to view how this version of *optimized-is-deallocated-at-higher-values* proceeds. Each stack in the diagram represents the search states from which the search needs to be re-run based on current knowledge about the search, along with their starting steps and the associated sensitivity value for agent 1 in that state. Below each stack we give the step of the actual search, along with the current value ($w_{1\ temp}$) for agent 1 and the current minimum next value ($w_{1\ min}$) to which sensitivity checking will jump once the current search is complete. $w_{1\ min}$ will always equal the value stored in the top search state in the stack. As we proceed from left to right, we see that we might add search states to the stack. This occurs if *get-sens-single-state* returns a next value that is lower than $w_{1\ min}$. Once we have run a search state to completion, we process the next search state in the stack, running *get-sensitivity* starting at the indicated step and jumping agent 1's value $w_{1\ temp}$ forward to the stored value. This is

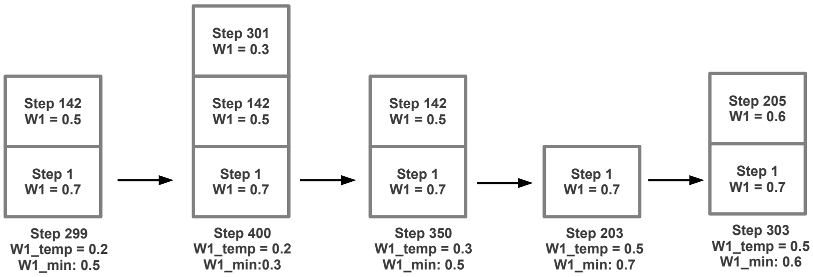


Fig. 5. Progression of *optimized-is-deallocated-at-higher-values*.

possible because we take a snapshot of the search state whenever we add a search state to the stack. The stack will expand and shrink, but the current agent value $w_{1\ temp}$ will monotonically increase, and eventually, we will have processed all search states in the stack and completed sensitivity analysis for the agent.

5.3.2. Inference: Allowing Early-Stopping. Until this point, we are still actually running the search to completion (even though hot restart lets us start low in the tree) for all higher values that trigger a sensitivity check, even though we only use the allocation to check whether the sensitivity agent remains in the allocation. This is all we care about: we don't need the full details of the allocation!

Leveraging this insight, we devise *early stopping rules* in the sensitivity checker. If we are sure that the search will terminate with a solution that contains agent 1, we do not need to run the search to completion (this is what we mean by "inference"). The main idea is to upper-bound the value of any solution where agent 1 is not allocated, and then use this upper bound to argue that the search will always terminate with a solution that allocates agent 1. One such upper bound is to take the max over the LP relaxations of all nodes in the current state, with the extra constraint $x_1 = 0$. At the cost of some extra computation (computing LP relaxations with the $x_1 = 0$), this allows us to stop searching once it is clear agent 1 will be allocated in any solution returned (see full version for more details).

5.4. Linear Program Caching, Parallelization

5.4.1. Linear Program Caching. The most expensive part of BnB search and sensitivity analysis is solving the LP relaxations for nodes. However, a key insight is that in the course of sensitivity analysis, we may revisit nodes with the same integer partial assignment over and over, with the only difference being that w_1 might be set to a higher value. As a result, when running *get-sensitivity*, we cache LP solutions, along with the upper bounds for when the LP solutions change (as in Section 5.1.2). When we need to solve an LP in a later BnB search with value w_1', we first make a lookup in this cache to see if there is an already computed LP solution whose upper bound is greater than w_1' and reuse the previously computed solution if one is found. This greatly decreases the number of solves needed for sensitivity analysis.[8]

5.4.2. Parallelization. While we have to perform sensitivity analysis for every allocated agent, the sensitivity analysis for each agent is completely independent of the sensitivity analysis for other agents. As a result, sensitivity checking can be perfectly

[8] An optimization related to LP caching is that of using optimal solutions from parent nodes in the BnB search tree to "hot start" the LP solve process for child nodes during sensitivity analysis. We did implement this, but we did not see substantial gains so we abandoned it for simplicity and to keep our memory footprint small. It would be of interest to pursue this direction further in future work.

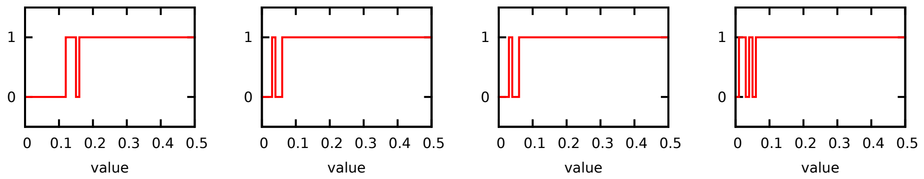


Fig. 6. The allocation function for different agents on a decay instance with 1000 agents and 100 items (see Section 7). Agents were chosen because of non-monotonicities in the BnB search for these agents.

parallelized. In our experimental results, we report this parallelized runtime, which is the time required to solve the initial search plus the maximum runtime for *optimized-is-deallocated-at-higher-values* across all allocated agents.

6. MAKING BRANCH-AND-BOUND SEARCH MORE MONOTONE

In order for the allocation computed by *discretized-ironed-alloc* to have good welfare properties, we need the underlying heuristic algorithm to be monotone for many agents on many instances. If not, then many agents will be deallocated, and even if the original, un-ironed solution has high welfare, the ironed solution will not. Recognizing this, we introduce two methods for making BnB more monotone.

Input Discretization: As discussed in Section 3, one way to decrease the number of deallocations is to increase the grid size β. With discretization, an agent remains allocated as long as the heuristic allocation function continues to allocate the agent for all higher multiples of β. Figure 6 shows the allocation curve for several agents in one of our experimental instances. The figure is generated using $\beta = 0.01$. Many of the non-monotonicities in the curves survive for a small range of values. Increasing β allows these small ranges to be skipped over and increases monotonicity. But there is a tradeoff with solution quality because the input is approximated.

Fractional Bucketing: The classic variable selection algorithm in BnB search is to take the most fractional variable; i.e., the variable with value closest to 0.5. But this is very sensitive to small changes in the LP solution, and can result in many search decision changes even if the selected node remains the same since the branch variable may change. To remedy this, we experiment with bucketing variables based on their fractionality and choosing the lexicographically first variable in the smallest bucket. For example, consider an LP solution $\{x_1 = 0.41, x_2 = 0.48, x_3 = 0.7, x_4 = 0.51\}$. The most fractional variable without any bucketing is x_4. But with a bucket size of 0.2, x_1, x_2, x_4 are all placed in the same bucket (the bucket representing values in $[0.4, 0.6]$), and we break ties on x_1. In the extreme case of a bucket size of 1.0, all variables belong to the same bucket, but we make the exception that we don't select variables that are already set to 0.0 or 1.0; therefore, a bucket size of 1.0 amounts to selecting the first variable that is set to a non-integer value. Larger bucket sizes make the underlying search more monotone since the decisions in the search are less sensitive to small changes in the LP solutions, and we see this in our experimental results.

7. EXPERIMENTAL RESULTS

We present experimental results based on an implementation of monotone BnB search for known single-minded CAs. Our experiments are performed using a custom Java implementation of BnB search, using CPLEX as our LP solver. The experiments are run on a machine with two 8 core 2.4GHz Intel Xeon processors. We implement the optimized version of *get-sensitivity*, as well as hot restart, inference / early-stopping, and LP caching.

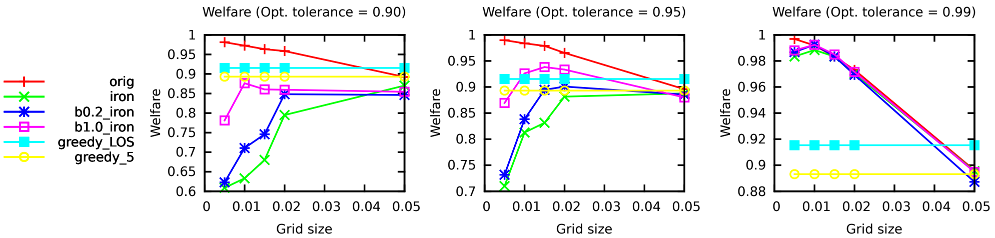


Fig. 7. Average welfare (compared to the optimal) for different search parameterizations on small instances.

We generate agent valuations using the *decay* (L4) distribution with parameter $\alpha = 0.75$ and a number of agents equal to ten times the number of items as this has been shown to generate hard winner determination instances [Leyton-Brown et al. 2000; Sandholm et al. 2005]. In our experiments, we fix node selection to choose the deepest node if no integral node has been found, and the node with highest value otherwise. For variable selection, we select the most fractional variable with different bucket sizes, as described in Section 6. For our discretization procedure, we first normalize values to $[0, 1]$ by dividing by a maximum value.[9]

7.1. Welfare Analysis

We generate 50 random instances from the Decay distribution with 300 agents, 30 items, and $\alpha = 0.75$. We vary γ, β, and the variable selection algorithm. For variable selection, the bucket sizes that we consider are *no bucket size*, 0.2 and 1.0. For this dataset, running to full optimality is very fast (on the order of seconds), so these instances do not represent a domain on which we would want to use our ironing procedure. Rather, they are a way to examine the impact of search parameterization on the quality of the ironed solution.

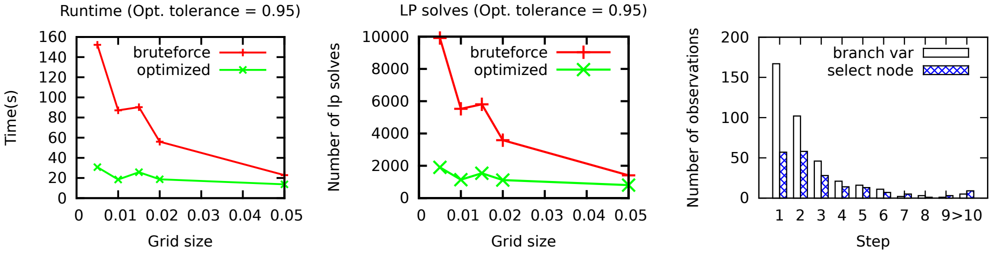
Figure 7 presents the welfare results. Each graph is for a particular $\gamma \in \{0.9, 0.05, 0.99\}$ and plots average welfare of the solution across the 50 instances (relative to the optimal) as β increases (i.e., more discretization.) *greedy-LOS* indicates the welfare of the greedy algorithm from Lehmann et al. [2002], while *greedy-5* indicates the welfare of the algorithm from Mu'alem and Nisan [2008] with a parameter choice of 5. Beyond this value, the runtime becomes prohibitive without much improvement in welfare (see full version).[10] The *orig* line indicates the *original welfare* of the solution, i.e. welfare before we check whether agents need to be deallocated. The *iron* line indicates the *ironed welfare*, i.e. welfare after agents have been deallocated.[11] Lines with b followed by a floating point number indicate use of bucket sizes. For instance, *b0.2-iron* plots the ironed welfare for bucket size 0.2.

7.1.1. Grid size (β). Figure 7 illustrates the effect of the grid size, β, on the welfare of the ironing algorithm. If β is too small, then there are many deallocations, and the ironed welfare suffers. If β is too large, then optimizing against the discretized values gives a poor approximation to the original problem, and welfare suffers.

[9] For the L4 distribution with $\alpha = 0.75$, 30 is a reasonable maximum value.

[10] The specific algorithm we compare against is the one that gives an $O(\epsilon\sqrt{m})$ approximation for single-minded CAs. For parameter k, this algorithm takes the max of exhaustive search over allocations with k agents and a greedy algorithm that uses a compact ranking, i.e. the score of a bundle T_i is w_i if $|T_i| \leq \sqrt{m/k}$ and 0 otherwise.

[11] To make the plots clearer, we plot the original welfare for the most fractional variable without bucketing. The original welfare is similar for the other bucketing strategies.



(a) (b)

Fig. 8. (a) Runtime and number of LP solves for brute force and optimized ironing procedures. (b) Histogram of the steps at which search decisions change, for different change types. $\gamma = 0.95, \beta = 0.01$, averaged over all bucket sizes and datasets. 300 agents, 30 items.

7.1.2. Most fractional bucket size. Figure 7 confirms that fractional-variable bucketing has a positive effect on the monotone BnB. The curves with the highest ironed welfare are those with bucket size 1.0.

7.1.3. Optimality tolerance (γ). The original welfare is quite similar across different values of γ. We also see that the welfare of the ironed solution improves as γ increases. In particular, for $\gamma > 0.95$, an optimally parameterized ironing algorithm does better than the greedy algorithms.[12] This occurs with $\gamma = 0.95$ despite monotonicity failure and agent deallocations.[13]

7.2. Effectiveness of optimized sensitivity

7.2.1. Comparison to brute force. We compare the brute force approach with the optimized sensitivity approach. We label the sensitivity checking procedure for BnB the “optimized” algorithm. Figure 8(a) plots the runtime and number of LPs solved across different grid sizes for $\gamma = 0.95$ and no bucket size.[14]

It shows that runtime is highly correlated with the number of LP solves. The optimized ironing procedure yields the biggest gains when the grid size is small, though for all grid sizes, the optimized procedure does have better runtime and a smaller number of calls to the LP solver. Brute force checks every higher multiple of the β and thus performs work linear in 1 / β. Even for large grid sizes, where the brute force procedure only needs to make a small number of calls to check sensitivity, the optimized procedure appears to match or slightly improve on its performance. This is likely due to the fact that the optimized procedure also leverages larger β in that it rounds to the next highest multiple of β when checking sensitivity.[15]

7.3. Analysis of search changes

We also study the particular types of decision changes that take place during sensitivity checking. For $\gamma = 0.95, \beta = 0.01$, most decision changes are branch variable changes rather than select node changes. With no bucket size, there is an average of 27.1 branch variable changes and 3.8 select node changes. For bucket size 1.0, these

[12] *greedy-LOS* and *greedy-5* do not have a grid size, so they appear as a constant line.

[13] The good performance is not because the underlying “orig” algorithm is identifying optimal solutions (and thus monotone). Averaged across all bucket sizes, for $\beta = 0.01$ and $\gamma = 0.9$, 34% of the original solutions are optimal (with respect to the particular grid size). This increases to 45% and 92% for $\gamma = 0.95$ and $\gamma = 0.99$.

[14] The graphs for different bucket sizes look very similar so for clarity, we only display no bucket size.

[15] We did not implement LP caching for the brute force trials, though, which could potentially decrease runtime and number of LP solves.

Table I. Welfare (% of optimal) on hard instances.

greedy (LOS)	orig-0.025-98-1.0	iron-0.025-98-1.0
0.94	0.96	0.96 (+0.02)
0.89	0.93	0.93 (+0.04)
0.92	0.93	0.93 (+0.00)
0.94	0.93	0.93 (-0.01)
0.87	0.92	0.92 (+0.05)
0.91	0.93	0.93 (+0.02)
0.89	0.93	0.92 (+0.03)
0.91	0.91	0.91 (+0.00)
0.88	0.93	0.93 (+0.04)
0.92	0.93	0.93 (+0.01)

Table II. Runtime (minutes) on hard instances

optimal	0.025-98-1.0 (t)	0.025-98-1.0 (p)
4.55	0.41	0.03
0.24	0.02	0.00
1.17	0.14	0.01
0.53	0.10	0.01
1.94	0.14	0.01
0.78	0.02	0.00
2.25	0.43	0.03
0.47	0.39	0.02
1.26	2.11	0.14
0.20	0.17	0.01

numbers decrease to 15.4 and 8.0, indicating that larger bucket sizes do decrease the number of branch variable changes. Figure 8(b) examines when these changes occur, and we see that branch variable changes tend to occur in the earlier steps of the search while select node changes are more evenly distributed.

7.4. Hard instances

In this section, we examine instances where optimal BnB is more computationally intensive and takes minutes to run to completion. We use decay instances with 1000 agents and 100 items, $\alpha = 0.75$. We test different parameters, but in Tables I and II, we focus on the best performing parameters $\beta = 0.025, \gamma = 0.98$ with bucket size 1.0. For welfare, $orig$ indicates the pre-ironed welfare, while $iron$ indicates the welfare after deallocations. For runtime, t indicates the total runtime for monotone BnB, while p indicates the fully parallelized runtime discussed in Section 5.4.2.

From Table I, we see that the welfare produced is better than greedy on these hard instances, and also that few agents are deallocated as the ironed welfare is close to the original welfare.[16] Table II gives the runtime for optimal BnB and the total and parallelized runtime for monotone BnB. Running to optimality tends to take more time than monotone BnB, but there are exceptions. In addition, the fully parallelized runtime (Section 5.4.2) for our algorithm is better than optimal BnB. We also note that to maintain truthfulness with optimal BnB we must be able to run every instance to completion, so we care about the long tail of the runtime distribution. With monotone BnB, the search itself is fast because we run to an optimality tolerance, and sensitivity checking for a single agent is not overly expensive. The expense comes in having to check every allocated agent, and as we have mentioned, this can be parallelized.[17]

8. FUTURE DIRECTIONS

We introduce a method for monotone BnB search by performing automated sensitivity analysis in regard to changes in the outcome of search in response to changes in objective value coefficients. We believe the results in regard to the scalability of sensitivity checking of BnB search are promising, and given the generality of the approach, hope to uncover additional optimizations. Possible areas for further improvement are additional inferential approaches that allow for short-circuiting, as well as additional ways to encourage monotonicity. We may also be able to leverage the fact that LP value, as a function of a specific agent's value, is convex.

Basic to our approach is the idea of performing sensitivity analysis for a given input and adjusting the algorithm *on that input* so that the algorithm is monotone over the

[16] We only report *greedy-LOS* since it outperforms parameterizations of *greedy-k* for values of k with runtimes comparable to monotone BnB.

[17] In this sense, we can reliably decrease runtime for monotone BnB with more computational resources, in contrast with the scalability that would be offered by state-of-the-art parallel branch-and-bound solvers.

entire input space. Probably the most intriguing, and challenging, direction for future work is to understand whether this local adjustment is possible in achieving appropriate notions of monotonicity in problems of multi-dimensional mechanism design. Additional targets for future work include: (1) Extend monotone BnB search to other single-dimensional mechanism design problems, including non-downward closed environments (e.g., scheduling, where correcting a failure of monotonicity could involve introducing additional “dummy” jobs for a machine to process); (2) Explore the idea of sensitivity and computational ironing on other methods of heuristic search, for example local search; (3) Extend monotone BnB search to handle cut generation, and expose a parameterized search framework to the methods of empirical algorithm design, to allow for automated configuration [Hutter et al. 2010]; and (4) Consider alternative methods to “correct” an allocation when a failure of monotonicity is identified, for example introducing randomization to allow for smoother notions of monotonicity.

Acknowledgments

We thank the anonymous reviewers for useful comments and feedback. This material is based upon work supported in part by the National Science Foundation under Grant No. CCF-1101570. John Lai is supported by an NDSEG fellowship.

REFERENCES

ANDERSSON, A., TENHUNEN, M., AND YGGE, F. 2000. Integer programming for combinatorial auction winner determination. In *Proc. of 4th ICMAS*. 39–46.

BERTSIMAS, D. AND TSITSIKLIS, J. 1997. *Introduction to Linear Optimization* 1st Ed. Athena Scientific.

CONSTANTIN, F. AND PARKES, D. C. 2009. Self-correcting sampling-based dynamic multi-unit auctions. In *Proc. of 10th ACM-EC Conference*. 89–98.

FEAUTRIER, P. 1988. Parametric integer programming. *RAIRO Recherche Op'erationnelle 22*.

HARTLINE, J. D., KLEINBERG, R., AND MALEKIAN, A. 2011. Bayesian incentive compatibility via matchings. In *Proc. of 22nd SODA*. 734–747.

HARTLINE, J. D. AND LUCIER, B. 2010. Bayesian algorithmic mechanism design. In *Proc. of 42nd STOC*. 301–310.

HUTTER, F., HOOS, H., AND LEYTON-BROWN, K. 2010. Automated configuration of mixed integer programming solvers. *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 186–202.

LAVI, R. AND SWAMY, C. 2005. Truthful and near-optimal mechanism design via linear programming. In *Proc. of 46th FOCS Symposium*. 595–604.

LEHMANN, D., O'CALLAGHAN, L. I., AND SHOHAM, Y. 2002. Truth revelation in approximately efficient combinatorial auctions. *Journal of the ACM 49*, 577–602.

LEYTON-BROWN, K., PEARSON, M., AND SHOHAM, Y. 2000. Towards a universal test suite for combinatorial auction algorithms. In *Proc. of 2nd ACM-EC Conference*. 66–76.

MARSTEN, R. AND MORIN, T. 1977. Parametric integer programming: The Right-Hand side case. *Annals of Discrete Mathematics 1*, 375–390.

MU'ALEM, A. AND NISAN, N. 2008. Truthful approximation mechanisms for restricted combinatorial auctions. *Games and Economics Behavior 64*, 612–631.

MYERSON, R. 1981. Optimal auction design. *Mathematics of Operations Research*, 58–73.

NEMHAUSER, G. AND WOLSEY, L. 1998. *Integer and Combinatorial Optimization*. J. Wiley and Sons, Inc.

PARKES, D. C. 2009. When analysis fails: Heuristic mechanism design via self-correcting procedures. In *Proc. of 35th SOFSEM*. 62–66.

PARKES, D. C. AND DUONG, Q. 2007. An ironing-based approach to adaptive online mechanism design in single-valued domains. In *Proc. of 22nd AAAI Conference*. 94–101.

PARKES, D. C. AND SCHOENEBECK, G. 2004. Growrange: Anytime vcg-based mechanisms. In *Proc. of 19th AAAI Conference*. 34–41.

SANDHOLM, T. 2002. Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence 135,* 1-2, 1–54.

SANDHOLM, T., SURI, S., GILPIN, A., AND LEVINE, D. 2005. Cabob: A fast optimal algorithm for winner determination in combinatorial auctions. *Management Science 51,* 3, 374–390.