Parallel Best-First Search for Planning Domains*

*Professor Didier El Baz (Dr. Eng., HDR)
Harbin Engineering University, China
(111 Program at Department of Computer Science of HEU)
University of Toulouse, France

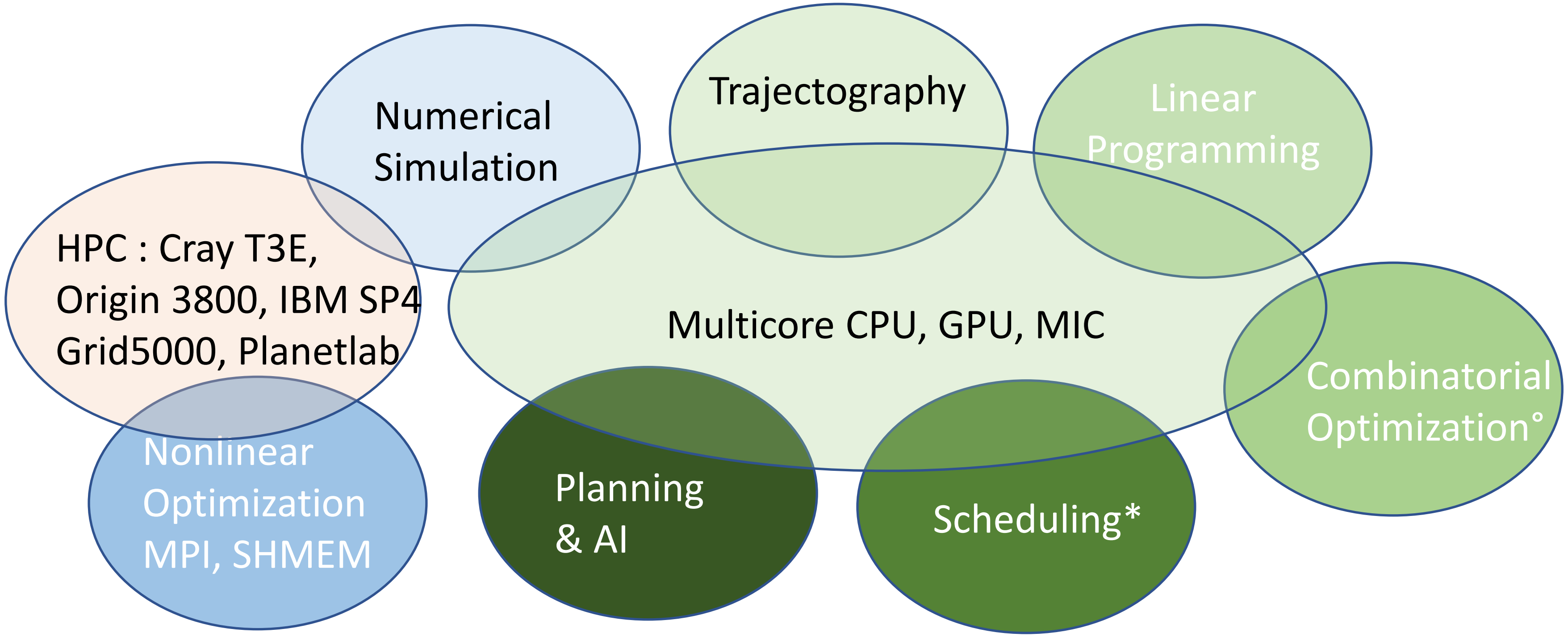
- D. El Baz, B. Fakih, R. Sanchez Nigenda, V. Boyer, Parallel best-first search algorithms for planning problems on multi-core processors, The Journal of Supercomputing

Outline

- 1. Introduction
- 2.Planning systems and best-first search
- 3. Parallel best-first search algorithms
- 4. Related work
- 5. Computational tests
- 6. Conclusions and future work

1. Introduction

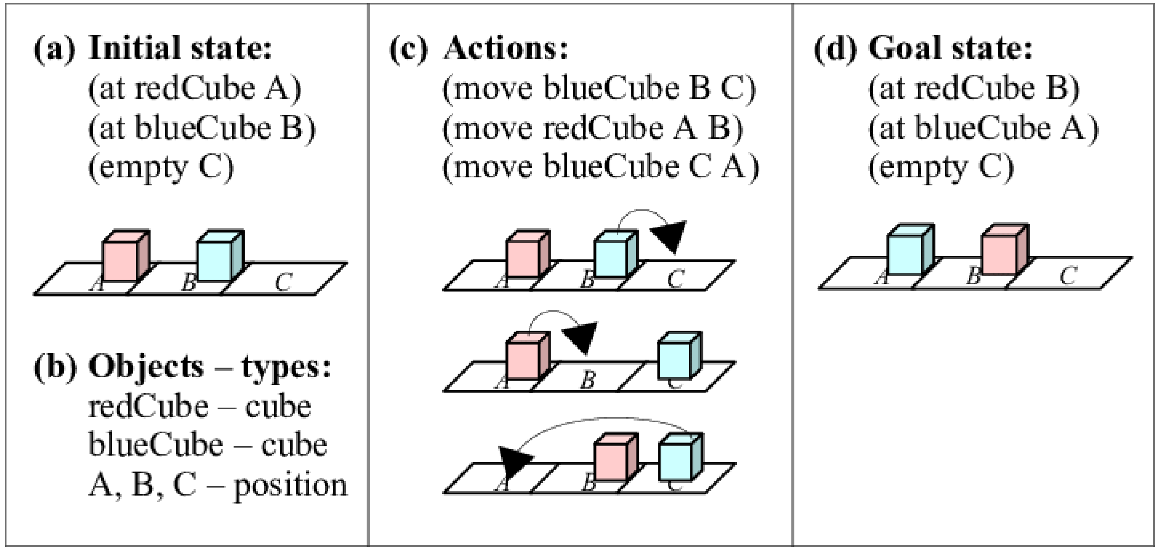
- **Fields of interest in Applied Maths, parallel computing and HPC**
- **First NVIDIA CUDA Tutorial in Europe, LAAS-CNRS University of Toulouse France 2008.**



* Journal of Parallel and Distributed Computing 2018, 2019, Fut. Gen. Comp. Syst 2020, An. of Op. Res. 2025
° JPDC 2005, COR 2012, IPDPS 2012.

1. Introduction

- AI Planning involves satisficing a goal state from an initial state through a series of action refinements.
  - ➢ Planning tasks for satellites, airport task planning, airline crew scheduling, autonomous robot navigation, DARS, intelligent transportation, and logistics…
- Complex problems.
- In general, planning algorithms solve problems through actions goal model representations, heuristics and metrics to control search.
  - ➢ Domain-independent planning methods
  - ➢ Best-first search (widely used in the context of planning).



1.1. Planning and parallel computing

- **Solution of AI planning problems via parallel heuristic methods.**
- **Progress of CPU:**

➢ **several dozens of computing cores;**

➢ **high memory bandwidth and efficient memory management;**

➢ **efficient task schedulers.**

➢ **Multi-core CPU and shared memory systems such as modern computing nodes with multi-core CPUs are excellent platforms that permit one to revisit approaches for solving planning problems.**

1.2. Parallel best-first search

- Parallel planning algorithms derived from best-first search are proposed for shared memory architectures. The parallel algorithms, based on the work pool paradigm, maintain good thread occupancy in multi-core CPUs.
- All algorithms use one ordered global list of states stored in shared memory from where they select nodes for expansion.
- A parallel best-first search algorithm that develops new states with depth equal to one is proposed first.
- An extension of this parallel algorithm that features a diversification strategy in order to escape local minima is also proposed.

1.3. Applications

- Various planning problems:

  Real world problems;
  International Planning Competition (IPC).

- Comparison with

➢ LPG-td,

➢ A*, ZHDA*, AZHDA*, FAZHDA*, OZHDA*, AHDA* DAHDA, GAZHDA* GRAZHDA * (Jinnai, Fukunaga).

2. Planning systems and best-first search

- Many of the most awarded planning systems base their implementations on best-first search derivatives introducing differences in how they compute heuristic estimates to guide the search process.

➢ Best-first search method on top of LPG-td.

✓ LPG-td fully-automated domain-independent planner for PDDL2.2 domains. It is based on best-first search and also stochastic local search and planning graphs.

✓ LPG-td won the best-automated planner award in the 2003 IPC and the best performance award in domains with timed literals in 2004.

2.1. Best-first search methods

- Best-first search is an instance of general graph and tree search algorithms that selects the next node for expansion based on the value of an evaluation function.

➢ The algorithm uses a priority queue to store the search nodes because it needs access to the best node, given the evaluation function, for expansion.

➢ A priority queue is usually implemented with **heaps**, a complex but efficient tree-based data structure that provides access to the object with the highest (or lowest) priority in $O(1)$ time.

➢ Once the best node is selected, each applicable operator, e.g., action, generates children nodes, which are inserted back into the priority queue, insertions take $O(\log n)$ time where n is the size of queue.

➢ The algorithm keeps selecting and expanding search nodes until a goal state is found.

2.2. Principle of best-first search

- Algorithm 1 summarizes best-first search.

➢ Algorithm 1 requires:

✓ the initial state i of the problem;

✓ goal state g that needs to be satisfied.

➢ The algorithm creates an empty priority queue q used to select nodes for expansion (initially, q only contains the initial state i).

➢ It creates also an empty table d of visited nodes which is checked for duplicates.

Algorithm 1: Best-First Search Algorithm

Input: The initial state i and goal state g of the problem

Output: A solution state s

```
PriorityQueue q;
Table d;
// First state of Global ordered list q
State s;
q.insert(i);
While True
    s = q.Remove();
    d.insert(s);
    Foreach child v of state s
        v_h = EVALFN(v, g);
        If v_h == 0
            return v;
        If v ∉ d (not a duplicate)
            q.insertorderly(v);
End
```

2.2. Principle of best-first search

- Algorithm 1 summarizes best-first search.

➢ Algorithm 1 requires:

✓ the initial state i of the problem;

✓ goal state g that needs to be satisfied.

➢ .The algorithm creates an empty priority queue q used to store nodes for expansion (initially, q only contains the initial state i).

➢ It creates also an empty table d of visited nodes which is checked for duplicates.

Algorithm 1: Best-First Search Algorithm

Input: The initial state i and goal state g of the problem

Output: A solution state s

```
PriorityQueue q;
Table d;
// First state of Global ordered list q
State s;
q.insert(i);
While True
    s = q.Remove();
    d.insert(s);
    Foreach child v of state s
        v_h = EVALFN(v, g);
        If v_h == 0
            return v;
        If v ∉ d (not a duplicate)
            q.insertorderly(v);
End
```

2.2. Principle of best-first search

- Algorithm 1 summarizes best-first search.
- ➢ Algorithm 1 requires:
- ✓ the initial state i of the problem;
- ✓ goal state g that needs to be satisfied.
- ➢ The algorithm creates an empty priority queue q used to select nodes for expansion (initially, q only contains the initial state i).
- ➢ It creates also an empty table d of visited nodes which is checked for duplicates detection.

Algorithm 1: Best-First Search Algorithm

Input: The initial state i and goal state g of the problem
Output: A solution state s

```
PriorityQueue q;
Table d;
// First state of Global ordered list q
State s;
q.insert(i);
While True
    s = q.Remove();
    d.insert(s);
    Foreach child v of state s
        v_h = EVALFN(v, g);
        If v_h == 0
            return v;
        If v ∉ d (not a duplicate)
            q.insertorderly(v);
End
```

2.2 Principle of best-first search

- .The main loop keeps removing the best node s from q until a child state that satisfies the goal is found.
- The current selected state s is inserted in table d to avoid revisiting it later during the search process.
- Algorithm 1 uses the available planning actions to generate children states v for s and inserts them in ascending order in q according to the value v_h of the evaluation function **EVALFN**(v, g), only if these children are not duplicates in d.

Algorithm 1: Best-First Search Algorithm

Input: The initial state i and goal state g of the problem
Output: A solution state s

```
PriorityQueue q;
Table d;
// First state of Global ordered list q
State s;
q.insert(i);
While True
    s = q.Remove();
    d.insert(s);
    Foreach child v of state s
        v_h = EVALFN(v, g);
        If v_h == 0
            return v;
        If v ∉ d (not a duplicate)
            q.insertorderly(v);
End
```

2.2 Principle of best-first search

- The main loop keeps removing the best node s from q until a child state that satisfies the goal is found.
- .The current selected state s is inserted in table d to avoid revisiting it later during the search process.
- Algorithm 1 uses the available planning actions to generate children states v for s and inserts them in ascending order in q according to the value v_h of the evaluation function **EVALFN**(v, g), only if these children are not duplicates in d.

Algorithm 1: Best-First Search Algorithm

Input: The initial state i and goal state g of the problem
Output: A solution state s

```
PriorityQueue q;
Table d;
// First state of Global ordered list q
State s;
q.insert(i);
While True
    s = q.Remove();
    d.insert(s);
    Foreach child v of state s
        v_h = EVALFN(v, g);
        If v_h == 0
            return v;
        If v ∉ d (not a duplicate)
            q.insertorderly(v);
End
```

2.2 Principle of best-first search

- The main loop keeps removing the best node s from q until a child state that satisfies the goal is found.
- The current selected state s is inserted in table d to avoid revisiting it later during the search process.
- Algorithm 1 uses the available planning actions to generate children states v for s and inserts them in ascending order in q according to the value v_h of the evaluation function $\textbf{EVALFN}(v, g)$, only if these children are not duplicates in d.

Algorithm 1: Best-First Search Algorithm

Input: The initial state i and goal state g of the problem
Output: A solution state s

```
PriorityQueue q;
Table d;
// First state of Global ordered list q
State s;
q.insert(i);
While True
    s = q.Remove();
    d.insert(s);
    Foreach child v of state s
        v_h = EVALFN(v, g);
        If v_h == 0
            return v;
        If v ∉ d (not a duplicate)
            q.insertorderly(v);
End
```

2.3. Evaluation function

- The evaluation function $\mathbf{EVALFN}(v,\ g)$ is given by a complex iterative procedure that evaluates the cost of future actions to reach goal state g from any state v during the search.
- Domain-independent planning systems use heuristics, computed from abstractions and relaxations of the original problem, to traverse their large search spaces.
- LPG-td best-first search procedure considers reachability information from relaxed temporal action graphs to weight the elements of the search space.

➢ Gerevini A, Saetti A, Serina I, Planning through stochastic local search and temporal action graphs in LPG. Artif. Intell. Res. 20: pp. 239 – 290, 2003.

3. Parallel best-first search algorithms

- Parallel solutions constructed on the top of LPG-td (we are bound to the heuristic estimates computed by LPG-td planner).
- Family of parallel asynchronous best-first search algorithms that leverage modern multi-core processors as well as computing nodes with shared memory architectures.
- Same data structure as sequential best-first search.

➢ The proposed parallel best-first search algorithms maintain a global ordered list q and a global table of states d.

✓ The global list q stores the set of states that have been generated but not yet expanded.

➢ The global table d stores the expanded states to detect possible duplications.

3.1. Principle of parallel best-first search methods

- **The parallel algorithms are based on the work pool paradigm.**
- Multi-threaded methods that generate as many threads as there are computing cores in the system, i.e., one thread per core.
- Threads expand states asynchronously from the ordered global list q.
- Maintain good thread occupancy in multi-core CPUs.
- Approach solves elegantly and efficiently multi-threaded computations in terms of execution time and memory.

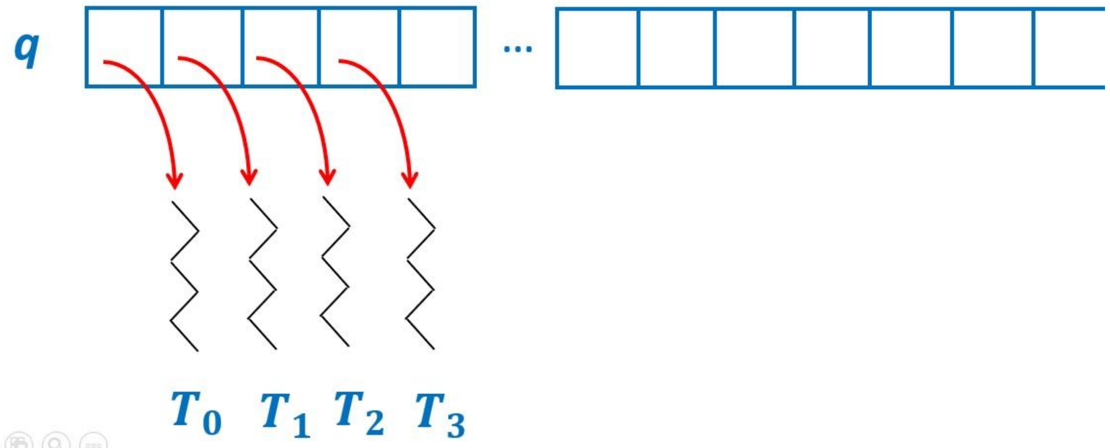


Fig. 3.1. Work pool paradigm,example with four threads

3.1. Principle of parallel best-first search methods

- When a thread T has no work, it retrieves a state from the global list q.
- When T generates a new state v for expansion, it places v in the global list q according to the value of the evaluation function if v is not a duplicate.

➢ The accesses to the ordered global list q are made via mutual exclusion techniques to avoid the simultaneous use of shared resources by different threads (maintaining data consistency and efficiency).

✓ The ordered list of states q is sometimes huge.

✓ A good thread occupancy is generally maintained in parallel multithreaded algorithms due to the complexity of the planning problems and large number of states to develop before finding a solution.

✓ Parallel threads access asynchronously, via mutual exclusion, both data structures q and d to control search.

3.2. First parallel best-first search method

- PBFSD1

➢ Each thread performs a best-first search with a depth equal to one.

➢ Newly created states, resulting from the best-first search procedure, are stored at one time in the ordered global list q via mutual exclusion if they are not duplicates.

3.3. First parallel best-first search algorithm

- .Expand the initial state i at the head processor after creating the empty global list and global table q and d, respectively.
- Thread T waits until the global list q is available and not locked by another thread.
- Thread T checks if a state is available from q. If so, then T retrieves in mutual exclusion the first state s of q, i.e., the state with the smallest value of evaluation function.
- Store the new state s in the global table d.
- Expand state s (produce one generation of children of state s).
- Thread T checks table d for each of the newly generated child v.
- If v is not a duplicate, then insert v in ascending order of the evaluation function in the list q.
- ✓ Writing operation is performed after obtaining a lock on the list q. This lock is released when the writing operation completes.
- If the value of the heuristic estimate of a child is equal to zero, then the algorithm reaches a solution state and terminates.

Algorithm 2: Parallel Best-First Search Algorithm *PBFSD1*

Input: The initial state i and goal state g of the problem
Output: A solution state s

```
Global PriorityQueue q;
Global Table d;
// initial phase
Foreach child v of state i
  v_h = EVALFN(v, g);
  If v_h == 0
    return v;
  q.insertorderly(v);
// First state of Global ordered list q
State s;
//Start parallel region
While True
  Mutual exclusion{
    s = q.Remove();
    d.insert(s);
  }
  // Produce one generation children v of state s
  Foreach child v of state s
    v_h = EVALFN(v, g);
    If v_h == 0
      return v;
    Mutual exclusion{
      If v ∉ d (not a duplicate)
        q.insertorderly(v);
    }
End
```

3.3. First parallel best-first search algorithm

- Expand the initial state i at the head processor after creating the empty global list and global table q and d, respectively.
- .Thread T waits until the global list q is available and not locked by another thread.
- .Thread T checks if a state is available from q. If so, then T retrieves in mutual exclusion the first state s of q, i.e., the state with the smallest value of evaluation function.
- Store the new state s in the global table d.
- Expand state s (produce one generation of children of state s).
- Thread T checks table d for each of the newly generated child v.
- If v is not a duplicate, then insert v in ascending order of the evaluation function in the list q.
- ✓ Writing operation is performed after obtaining a lock on the list q. This lock is released when the writing operation completes.
- If the value of the heuristic estimate of a child is equal to zero, then the algorithm reaches a solution state and terminates.

Algorithm 2: Parallel Best-First Search Algorithm *PBFSD1*

Input: The initial state i and goal state g of the problem
Output: A solution state s

```
Global PriorityQueue q;
Global Table d;
// initial phase
Foreach child v of state i
   v_h = EVALFN(v, g);
   If v_h == 0
      return v;
   q.insertorderly(v);
// First state of Global ordered list q
State s;
//Start parallel region
While True
   Mutual exclusion{
      s = q.Remove();
      d.insert(s);
   }
   // Produce one generation children v of state s
   Foreach child v of state s
      v_h = EVALFN(v, g);
      If v_h == 0
         return v;
      Mutual exclusion{
         If v ∉ d (not a duplicate)
            q.insertorderly(v);
      }
End
```

3.3. First parallel best-first search algorithm

- Expand the initial state i at the head processor after creating the empty global list and global table q and d, respectively.
- Thread T waits until the global list q is available and not locked by another thread.
- Thread T checks if a state is available from q. If so, then T retrieves in mutual exclusion the first state s of q, i.e., the state with the smallest value of evaluation function.
- Store the new state s in the global table d.
- Expand state s (produce one generation of children of state s).
- Thread T checks table d for each of the newly generated child v.
- If v is not a duplicate, then insert v in ascending order of the evaluation function in the list q.
- ✓ Writing operation is performed after obtaining a lock on the list q. This lock is released when the writing operation completes.
- If the value of the heuristic estimate of a child is equal to zero, then the algorithm reaches a solution state and terminates.

Algorithm 2: Parallel Best-First Search Algorithm *PBFSD1*

Input: The initial state i and goal state g of the problem
Output: A solution state s

```
Global PriorityQueue q;
Global Table d;
// initial phase
Foreach child v of state i
   v_h = EVALFN(v, g);
   If v_h == 0
      return v;
   q.insertorderly(v);
// First state of Global ordered list q
State s;
//Start parallel region
While True
   Mutual exclusion{
      s = q.Remove();
      d.insert(s);
   }
   // Produce one generation children v of state s
   Foreach child v of state s
      v_h = EVALFN(v, g);
      If v_h == 0
         return v;
      Mutual exclusion{
         If v ∉ d (not a duplicate)
            q.insertorderly(v);
      }
End
```

3.3. First parallel best-first search algorithm

- Expand the initial state i at the head processor after creating the empty global list and global table q and d, respectively.
- Thread T waits until the global list q is available and not locked by another thread.
- Thread T checks if a state is available from q. If so, then T retrieves in mutual exclusion the first state s of q, i.e., the state with the smallest value of evaluation function.
- Store the new state s in the global table d.
- Expand state s (produce one generation of children of state s).
- Thread T checks table d for each of the newly generated child v.
- If v is not a duplicate, then insert v in ascending order of the evaluation function in the list q.

✓ Writing operation is performed after obtaining a lock on the list q. This lock is released when the writing operation completes.

- If the value of the heuristic estimate of a child is equal to zero, then the algorithm reaches a solution state and terminates

Algorithm 2: Parallel Best-First Search Algorithm *PBFSD1*

```
Input: The initial state i and goal state g of the problem
Output: A solution state s
Global PriorityQueue q;
Global Table d;
// initial phase
Foreach child v of state i
   v_h = EVALFN(v, g);
   If v_h == 0
      return v;
   q.insertorderly(v);
// First state of Global ordered list q
State s;
//Start parallel region
While True
   Mutual exclusion{
      s = q.Remove();
      d.insert(s);
   }
   // Produce one generation children v of state s
   Foreach child v of state s
      v_h = EVALFN(v, g);
      If v_h == 0
         return v;
      Mutual exclusion{
         If v ∉ d (not a duplicate)
            q.insertorderly(v);
      }
End
```

3.3. First parallel best-first search algorithm

- Expand the initial state i at the head processor after creating the empty global list and global table q and d, respectively.
- Thread T waits until the global list q is available and not locked by another thread.
- Thread T checks if a state is available from q. If so, then T retrieves in mutual exclusion the first state s of q, i.e., the state with the smallest value of evaluation function.
- Store the new state s in the global table d.
- Expand state s (produce one generation of children of state s).
- Thread T checks table d for each of the newly generated child v.
- If v is not a duplicate, then insert v in ascending order of the evaluation function in the list q.
- ✓ Writing operation is performed after obtaining a lock on the list q. This lock is released when the writing operation completes.
- If the value of the heuristic estimate of a child is equal to zero, then the algorithm reaches a solution state and terminates.

Algorithm 2: Parallel Best-First Search Algorithm *PBFSD1*

Input: The initial state i and goal state g of the problem
Output: A solution state s

```
Global PriorityQueue q;
Global Table d;
// initial phase
Foreach child v of state i
   v_h = EVALFN(v, g);
   If v_h == 0
      return v;
   q.insertorderly(v);
// First state of Global ordered list q
State s;
//Start parallel region
While True
   Mutual exclusion{
      s = q.Remove();
      d.insert(s);
   }
   // Produce one generation children v of state s
   Foreach child v of state s
      v_h = EVALFN(v, g);
      If v_h == 0
         return v;
      Mutual exclusion{
         If v ∉ d (not a duplicate)
            q.insertorderly(v);
      }
End
```

3.3. First parallel best-first search algorithm

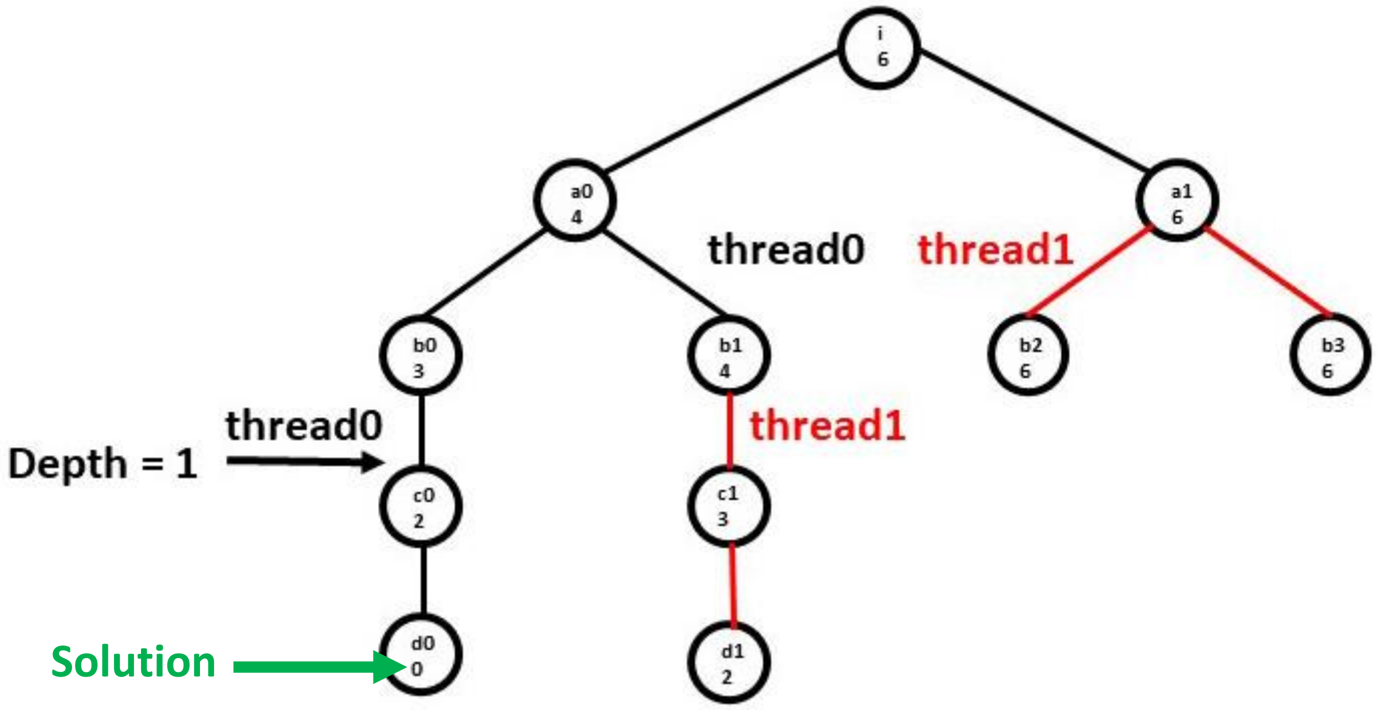


Fig. 3.2. Simple case of first parallel best-first search method PBFSD1

3.4. Second parallel best-first search method

- One of the effects of heuristics is that they might focus on local solutions.

➢ In other words, search exploration using parallel threads might not be diverse enough.

- The proposed parallel algorithm *PBFSD1* always picks nodes from the top of list q (best nodes given the heuristics).
- This strategy, which seems reasonable, might not be sufficient if the heuristic estimates are conservative.
- We propose the **Parallel Search** (***PS***) algorithm that performs a best-first search with **diversification**.

3.5. Principle of second parallel method

- Asynchronous Work Pool (AWP) paradigm and list of states q and table d.
- **Parallel algorithm *PS* is a combination of best-first search and diversification.**
  - **The multiple threads of *PS* algorithm perform randomly either a best-first search with a depth equal to one.**
  - **or they develop a state situated at 30% of list q; the selected state is then expanded along twenty generations according to best-first search principle (determined empirically).**

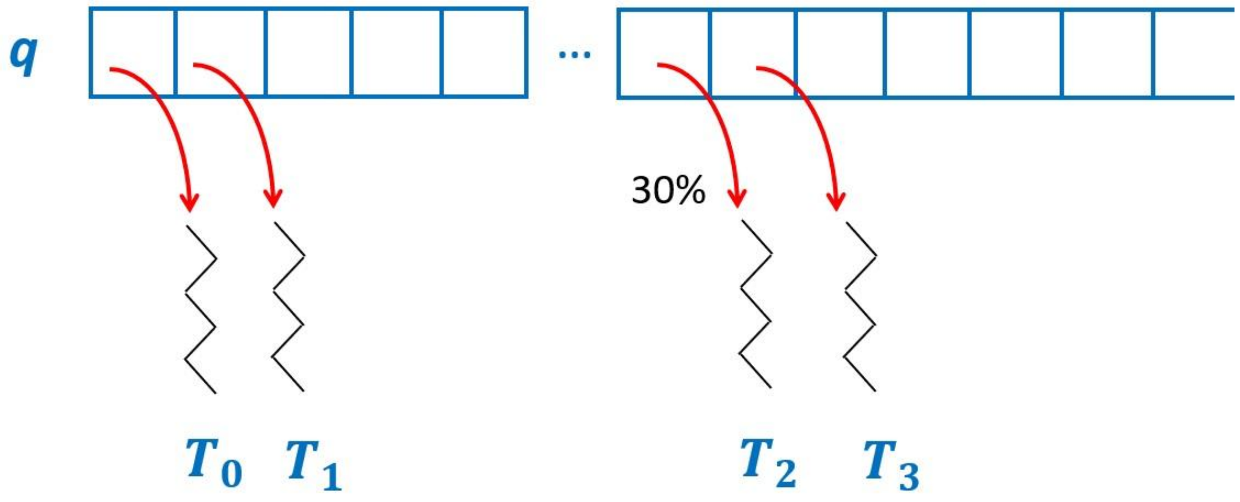


Fig. 3.3 Principle of algorithm PS, WP paradigm, example with four threads

3.5. Principle of second parallel method

- Global ordered list of states q and table d of visited states.
- **Picking by random states that are not the best in the ordered list and developing these states during a convenient number of generations according to best-first search principle has the potential to generate a new best state.**
- The probabilities to perform either a best-first search with depth equal to one from the best state or a best-first search with depth equal to twenty from a state situated at thirty percent of the global list *q* are identical and set equal to 0.5.
- **Algorithm *PS* is also implemented according to the work pool parallel paradigm.**

3.6. Second parallel best-first search algorithm

- Expansion of the initial state *i* at the head processor.

- Parallel region.

Algorithm 3: Parallel Search Algorithm *PS*

Input: The initial state *t* and goal state *g* of the problem
Output: A solution state *s*

```
Global PriorityQueue q;
Global Table d;
// Initial phase
Foreach child v of state t
   v_h = EVALFN(v, g);
   If v_h == 0
      return v;
   q.insertorderly(v);
// First state of Global ordered list q
State s;
// State at 30% of Global ordered list q
State t;
//Start parallel region
While True
   x_random := generateRandomInteger[1, 100]
   If x_random < 50{
      Mutual exclusion{
         s = q.Remove();
         d.insert(s);
      }
      // Produce one generation children v of state s
      Foreach child v of state s
         v_h = EVALFN(v, g);
         If v_h == 0
            return v;
         Mutual exclusion{
            If v ∉ d (not a duplicate)
               q.insertorderly(v);
         }
   }
   else{
      Mutual exclusion{
         // Point to state t at 30% of the global list q
         t = q.Remove();
         d.insert(t);
      }
      //Produce twenty generations descendants v of state t according to
   best-first search
      Foreach v
         If EVALFN(v, g) == 0
            return v;
         Mutual exclusion{
            If v ∉ d (not a duplicate)
               q.insertorderly(v);
         }
   }
End
```

3.6. Second parallel best-first search algorithm

- Expansion of the initial state *i* at the head processor.

- .Parallel region.

Algorithm 3: Parallel Search Algorithm *PS*

```
Input: The initial state i and goal state g of the problem
Output: A solution state s
Global PriorityQueue q;
Global Table d;
// Initial phase
Foreach child v of state i
   v_h = EVALFN(v, g);
   If v_h == 0
      return v;
   q.insertorderly(v);
// First state of Global ordered list q
State s;
// State at 30% of Global ordered list q
State t;
//Start parallel region
While True
   x_random := generateRandomInteger[1, 100]
   If x_random < 50{
      Mutual exclusion{
         s = q.Remove();
         d.insert(s);
      }
      // Produce one generation children v of state s
      Foreach child v of state s
         v_h = EVALFN(v, g);
         If v_h == 0
            return v;
         Mutual exclusion{
            If v ∉ d (not a duplicate)
               q.insertorderly(v);
         }
   }
   else{
      Mutual exclusion{
         // Point to state t at 30% of the global list q
         t = q.Remove();
         d.insert(t);
      }
      //Produce twenty generations descendants v of state t according to best-first search
      Foreach v
         If EVALFN(v, g) == 0
            return v;
         Mutual exclusion{
            If v ∉ d (not a duplicate)
               q.insertorderly(v);
         }
   }
End
```

3.6. Second parallel best-first search algorithm

- Develop either one generation children starting from state s at the beginning of list q.

```
//Start parallel region
While True
  x_random := generateRandomInteger[1, 100]
  If x_random < 50{
    Mutual exclusion{
      s = q.Remove();
      d.insert(s);
    }
    // Produce one generation children v of state s
    Foreach child v of state s
      v_h = EVALFN(v, g);
      If v_h == 0
        return v;
      Mutual exclusion{
        If v ∉ d (not a duplicate)
          q.insertorderly(v);
      }
}
```

3.6. Second parallel best-first search algorithm

- Develop twenty generations of children according to best-first search scheme starting from a state *t*, that is situated at thirty percent of the global ordered list.

```
  else{
    Mutual exclusion{
      // Point to state t at 30% of the global list q
      t = q.Remove();
      d.insert(t);
    }
    //Produce twenty generations descendants v of state t according to
best-first search
    Foreach v
      If EVALFN(v, g) == 0
        return v;
      Mutual exclusion{
        If v ∉ d (not a duplicate)
          q.insertorderly(v);
      }
  }
End
```

3.6. Second parallel best-first search algorithm

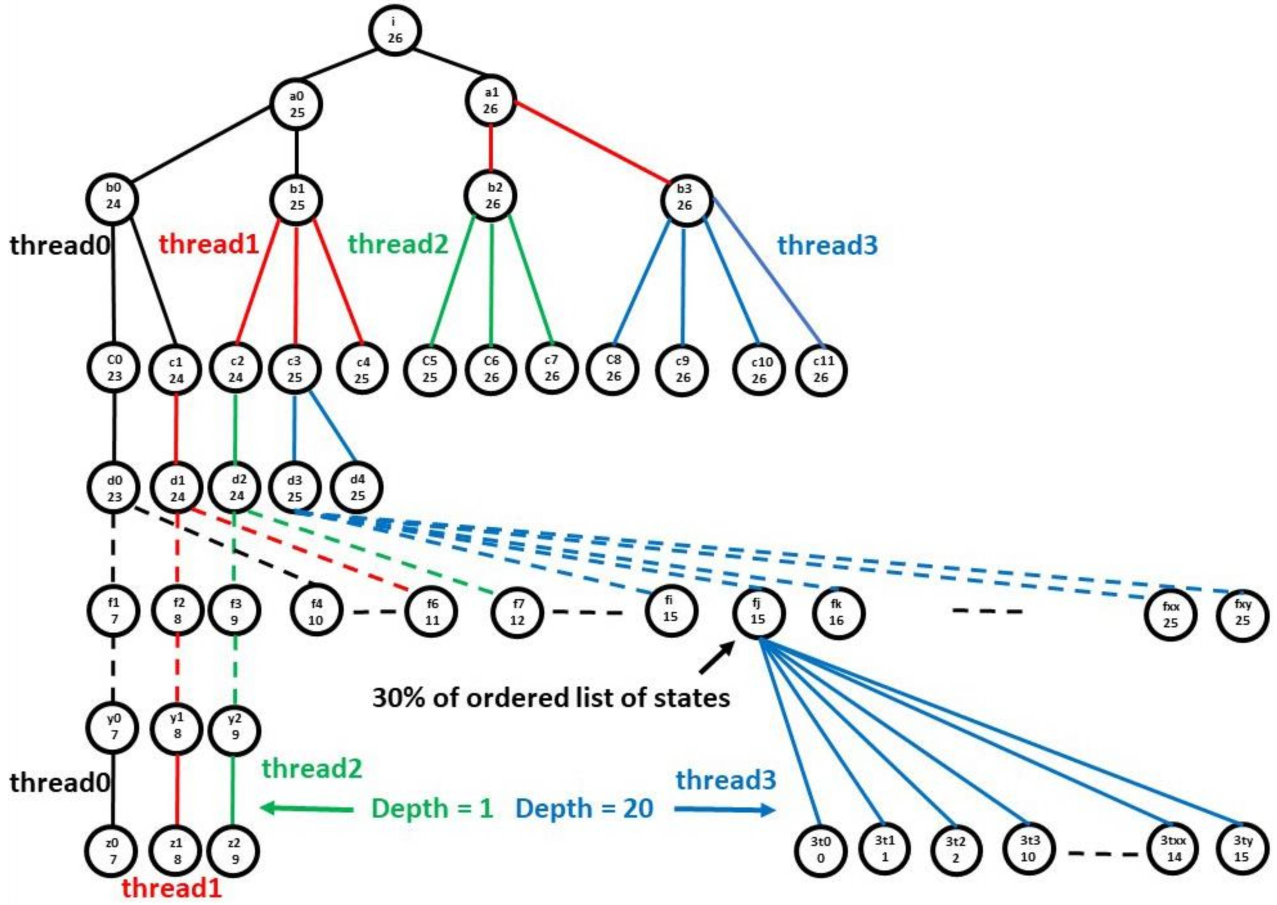


Fig. 3.4 Simple case of the second parallel best-first search method PS

4. Related work

- Parallel Retracting A* (PRA*), parallel implementation of RA* on Connection Machine.

➢ Evett M, Hendler J, Mahanti A, Nau D, (1995)

✓ PRA* distributes work among processors using a state hash function.

✓ Hash function maps each state generated to a corresponding processor.

✓ Each processor maintains its own open and closed lists (local lists).

✓ Open list stores the states that have been generated but not yet expanded.

✓ Closed list keeps the expanded states to detect duplicates.

➢ PRA* has a significant synchronization overhead since some processors have to wait for others to reach the synchronization point.

✓ When processor P generates and sends a new state to processor R, P is blocked until it receives a confirmation message from R. This mechanism is required since PRA* is implemented on a processor with a limited amount of local memory.

✓ PRA* uses a retraction mechanism to remove nodes from memory when needed.

4. Related work

- Hash-Distributed A* (HDA*) is a distributed implementation of the A* algorithm which asynchronously distributes and schedules work among processors based on a hash function of the search state.

➢ Kishimoto A, Fukunaga A, Botea A (2013)

✓ In particular, HDA* is an algorithm that combines the hash-based work distribution strategy of PRA* and the asynchronous communication of TDS.

➢ TDS distributes a transposition table among processors instead of open and closed lists

✓ HDA* is implemented on top of the Fast Downward domain-independent planner.

✓ As opposed to PRA*, HDA* does not incorporate a node retraction mechanism.

4. Related work

- More recent works on HDA* have focused on improving the hash function of HDA* that asynchronously distributes work.

➢ They concentrate on increasing the speedups of the HDA* algorithm by reducing node transfers and by mitigating communication overhead using abstract Zobrist hashing methods.

➢ **Implemented on distributed memory architectures (message passing), e.g., clusters, cloud.**

HDA*, ZHDA*, FAZHDA*, OZHDA*, AHDA* DAHDA, GRAZHDA GAZHDA* GRAZHDA *.

➢ Jinnai Y, Fukunaga A (2016), Jinnai Y, Fukunaga A (2017);

➢ Kuroiwa R, Fukunaga A (2019)

4. Related work

- An adaptive K-parallel best-first search algorithm, designed specifically for multi-core domain independent planning implemented on the top of the YAHSP planning system.

➢ Vidal V, Bordeaux L, Hamadi Y (2010).

- Parallel best-first search methods, implemented on shared or distributed memory architectures.

➢ Burns E, Lemons S, Ruml W, Zhou R (2010).

- **Parallel Best-first search algorithms that are suited for both multi-core and multi-machine clusters have not been previously evaluated in depth.**

5. Computational tests

- Various planning problems

➢ 824 problem instances in total for evaluation from 11 different planning domains.

✓ Real world problems;

✓ International Planning Competition (IPC).

5. Computational tests

- Computing node with **two Intel CPUs** Xeon Gold 6130, with 16 cores, clock 2.10 GHz, and 192 GB of RAM (product collection: Intel Xeon Scalable Processors).

➢**Total of 32 computing cores**.

✓OpenMP.

✓We do not pin threads to given cores;

❖Scheduler of CPUs assign threads to cores; assignment can change during a run.

5.1. Global results

- LPG-td sequential algorithm solved 687 problems from the evaluation set (83%).

➢ PBFSD1 solved 732 problems (88%);

➢ PS solved 737 problems (89%).

5.2. Real world

- three different planning domains from real-world applications.

➢ The first couple of problems come from manufacturing production plants.

✓ Single machine scheduling scenarios that consider maintenance operations.

✓ Maintenance operations have sequence-dependent setup costs that have to respect the availability constraints of the production line.

❖ *1) Model-EOL* (*End of Line*), each planning horizon must end with maintenance job.

❖ 2) *Model-ExE* , such tasks are *External Events* in the production plan.

➢ Third planning domain solves Public Transportation Network problems (*Model-PTN*).

✓ Engineering of travel plans taking into account the public transport and user preferences.

5.2.1. Model-EOL problems

each planning horizon must end with maintenance job.

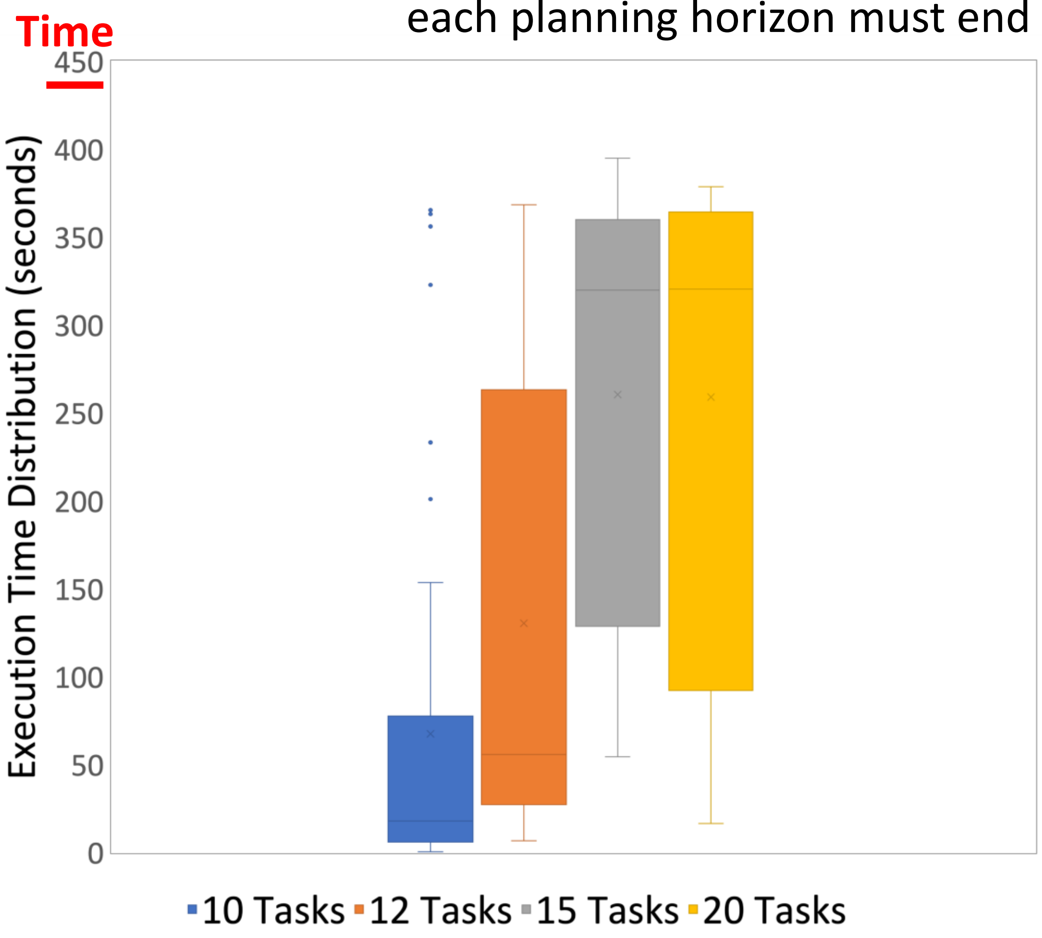


Fig. 5.1. LPG-td Execution time for 10 to 20 tasks

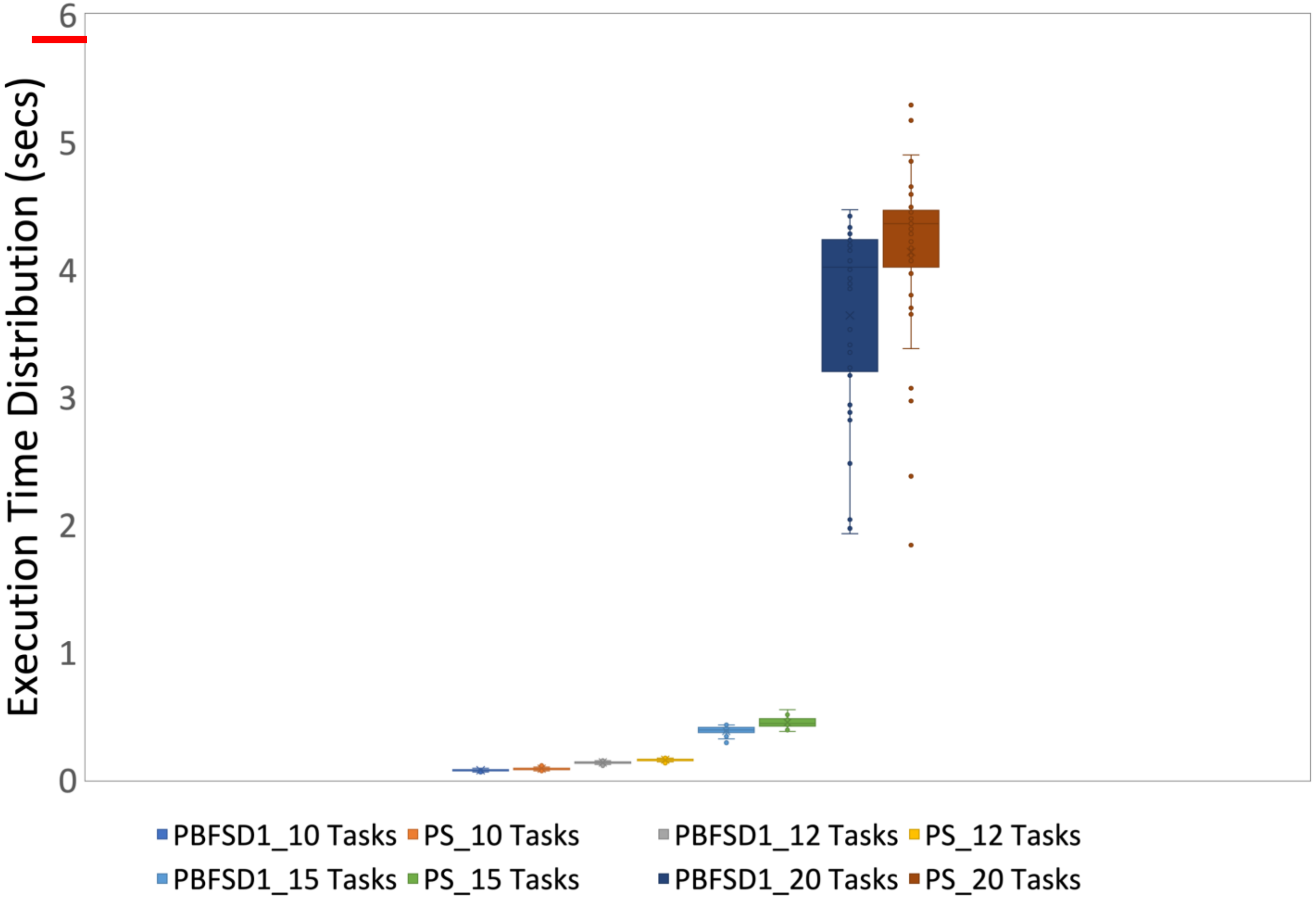


Fig. 5.2. PBFSD1 and PS Execution time for 10 to 20 tasks

5.2.1. Model-EOL problems

- PBFSD1 and PS solve all instances with 30 tasks.
- LPG-td does not solve any problem with 30 tasks.

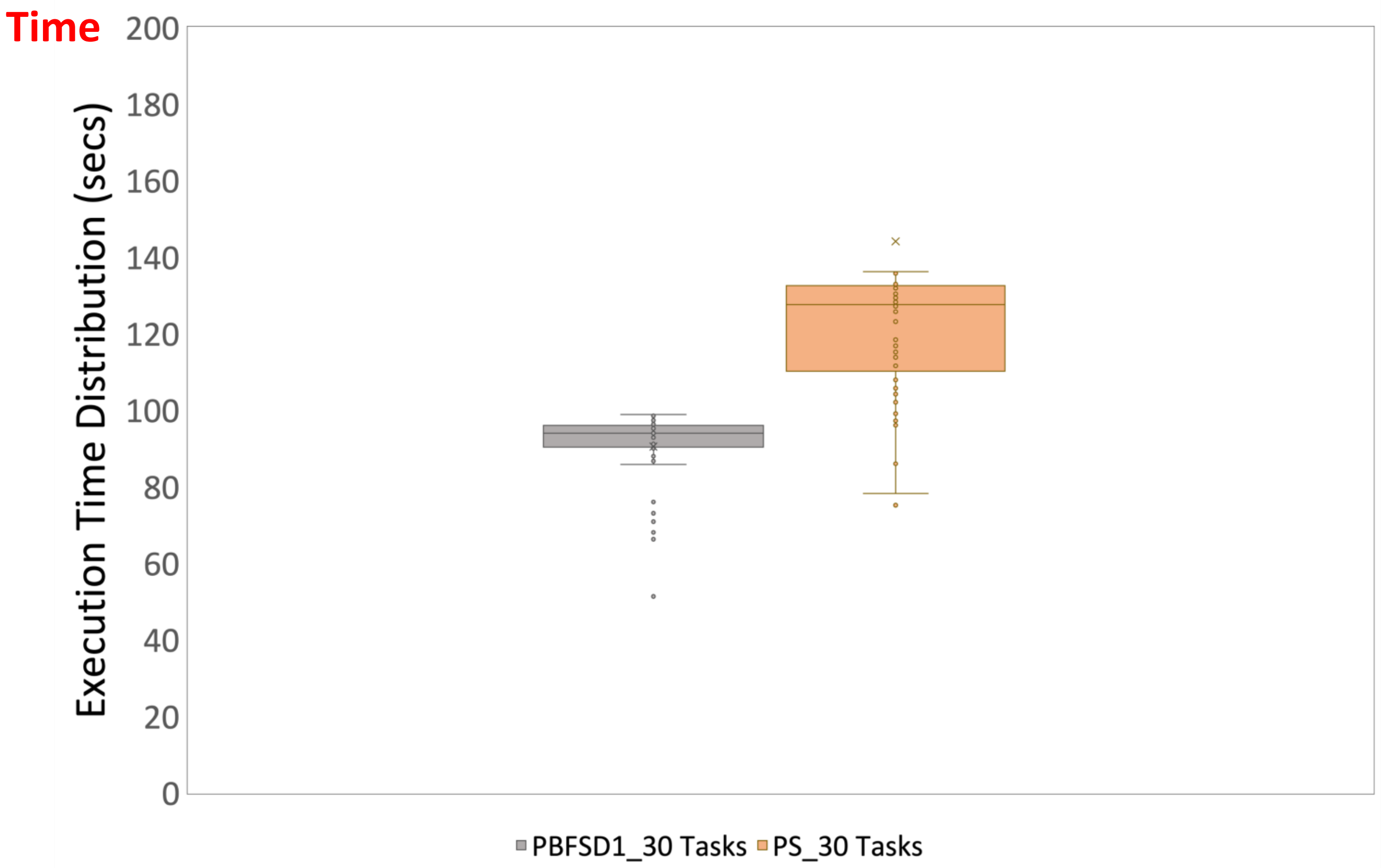


Fig. 5.3. PBFSD1 and PS Execution time for 30 tasks

5.2.1. Model-EOL problems

Quality of solutions

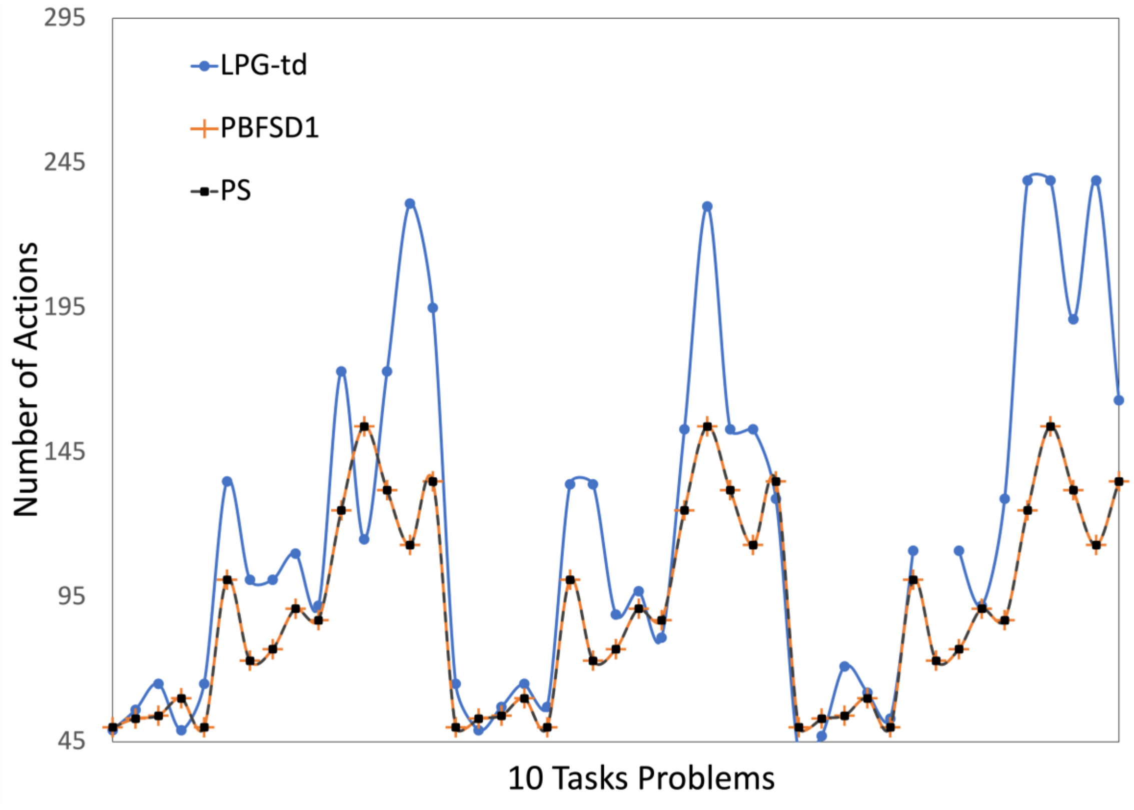


Fig. 5.4. Number of actions for problems with 10 tasks

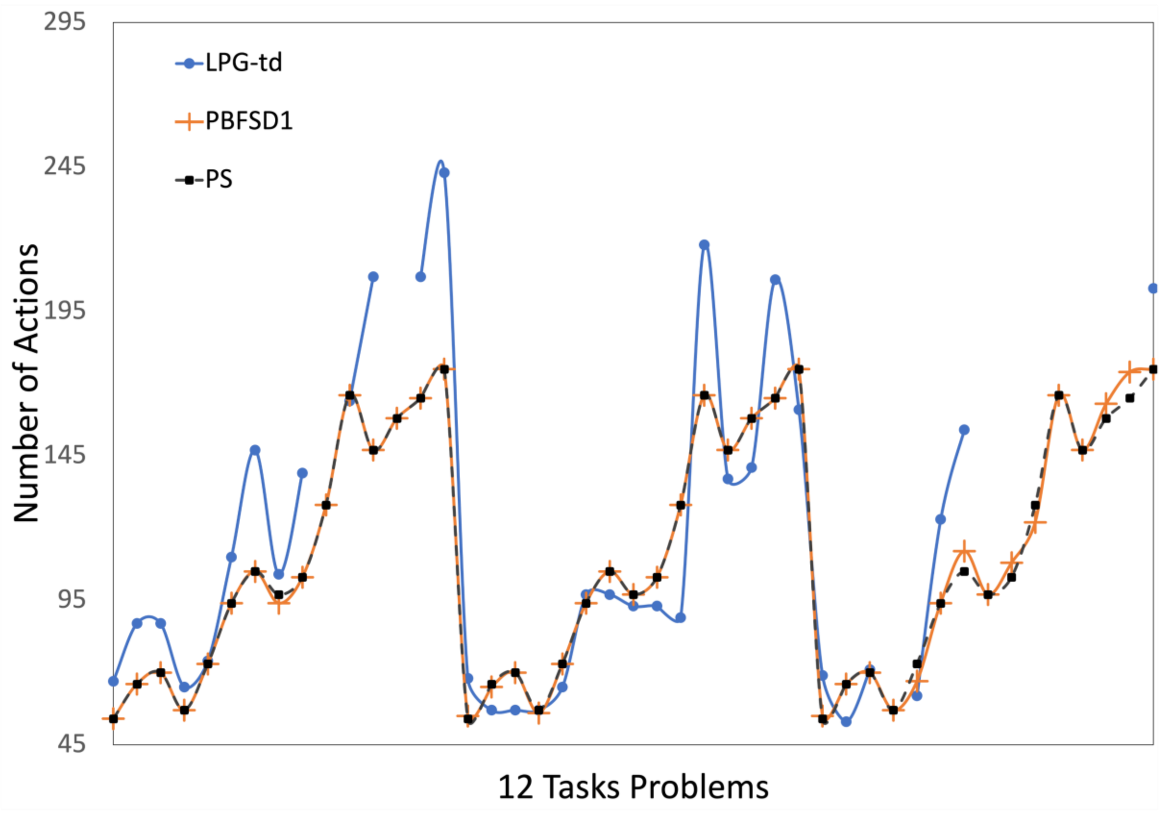


Fig. 5.5. Number of actions for problems with 12 tasks

5.2.1. Model-EOL problems

Quality of solutions

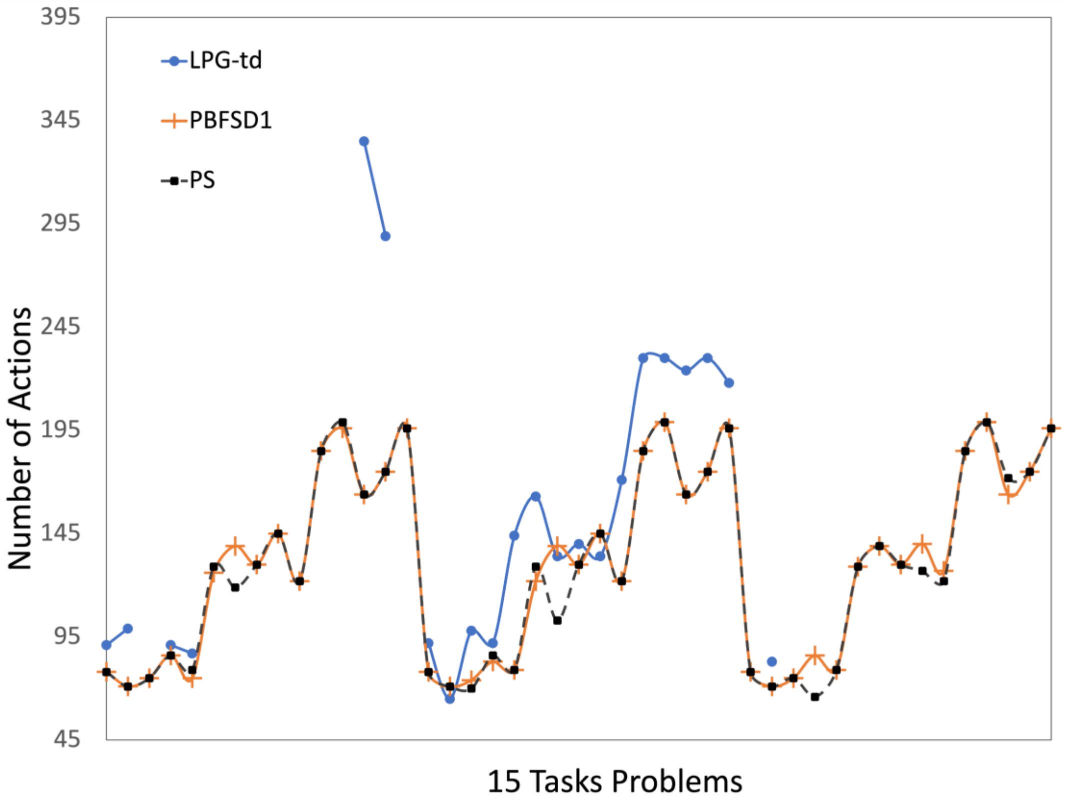


Fig. 5.6. Number of actions for problems with 15 tasks

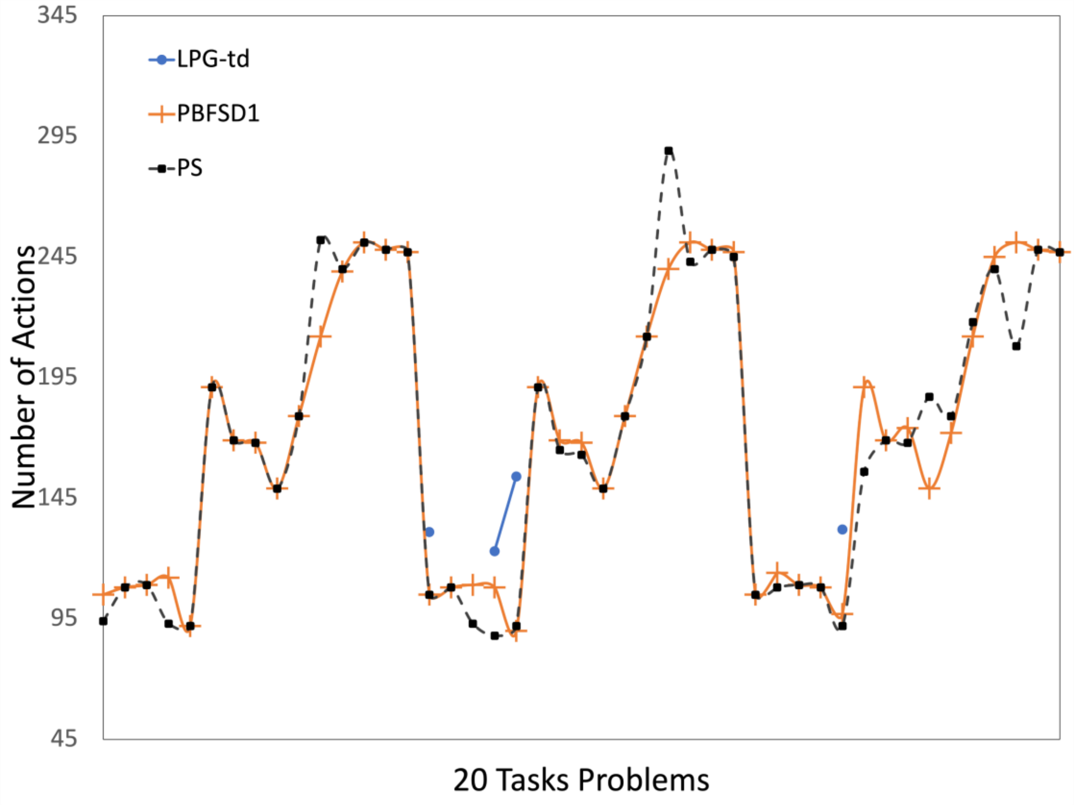


Fig. 5.7. Number of actions for problems with 20 tasks

5.2.1. Model-EOL problems

Quality of solutions

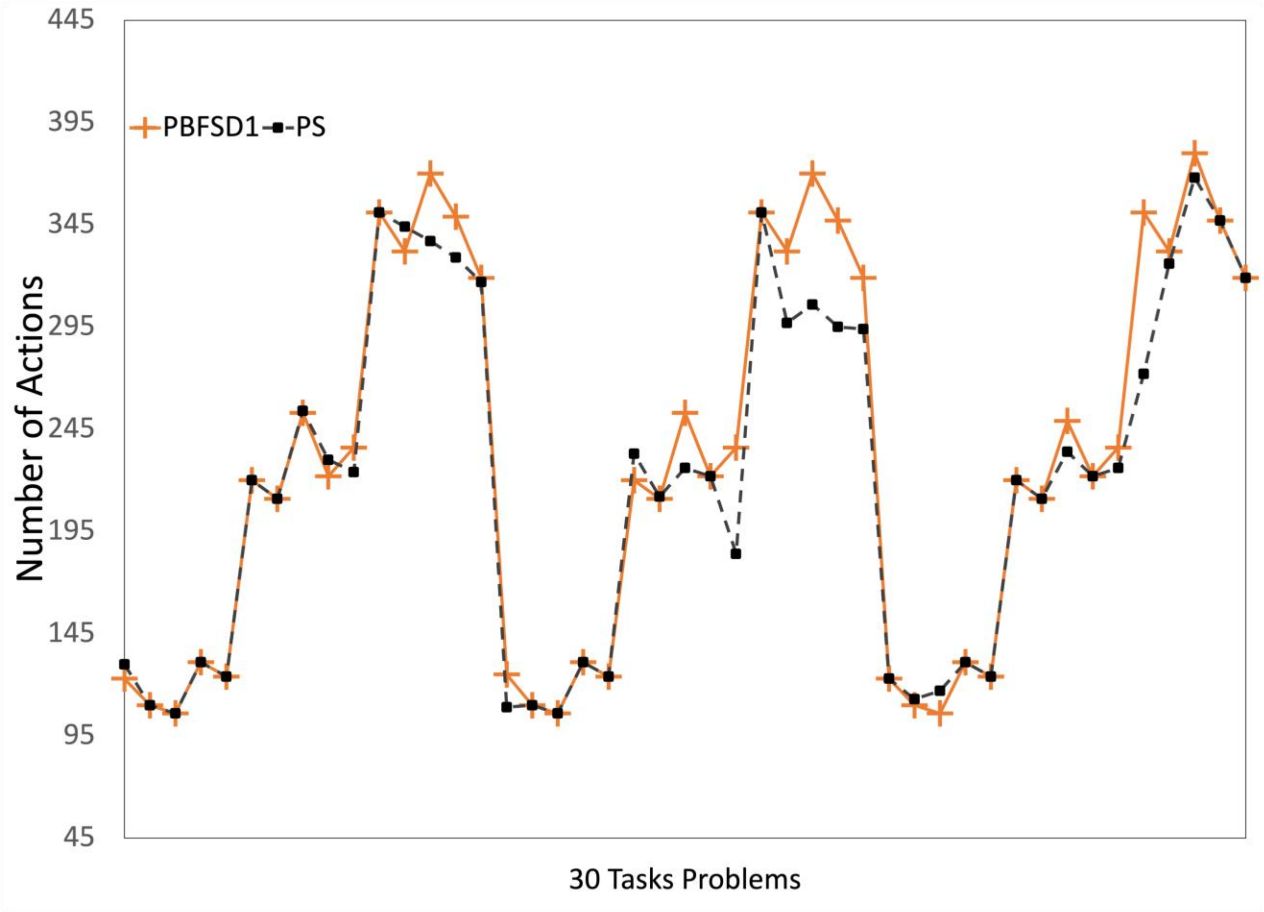


Fig. 5.8 Number of actions for problems with 30 tasks

5.2.1. Model-EOL problems

- 225 instances in total.

➢ *PBFSD1* and *PS* return solutions to every problem; LPG-td solves 46% of problems.

✓LPG-td does not scale up to problems with 30 tasks.

✓For the simplest problem, LPG-td generates longer plans on average in 28 cases.

➢ *PS* returns slightly shorter plans on average than PBFSD1.

✓*PBFSD1 and PS are faster than LPG-td in 99% of cases.*

➢ *PS* is slightly slower than *PBFSD1* on average.

➢ *PBFSD1* is 37% faster than *PS f*or problems with 30 tasks for the largest scenarios.

5.2.2. Model-ExE problems

maintenance jobs are *External Events* in the production plan.

Time

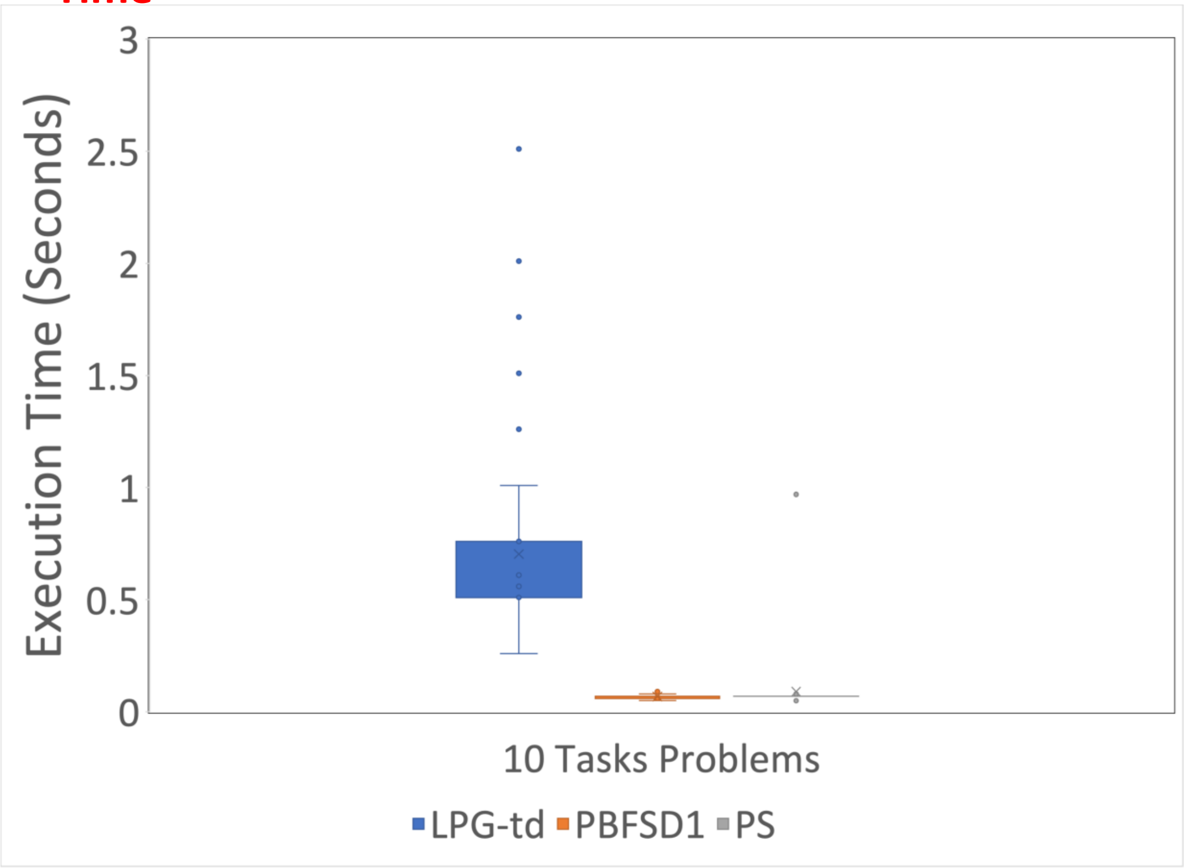


Fig. 5.9. Execution time for problems with 10 tasks

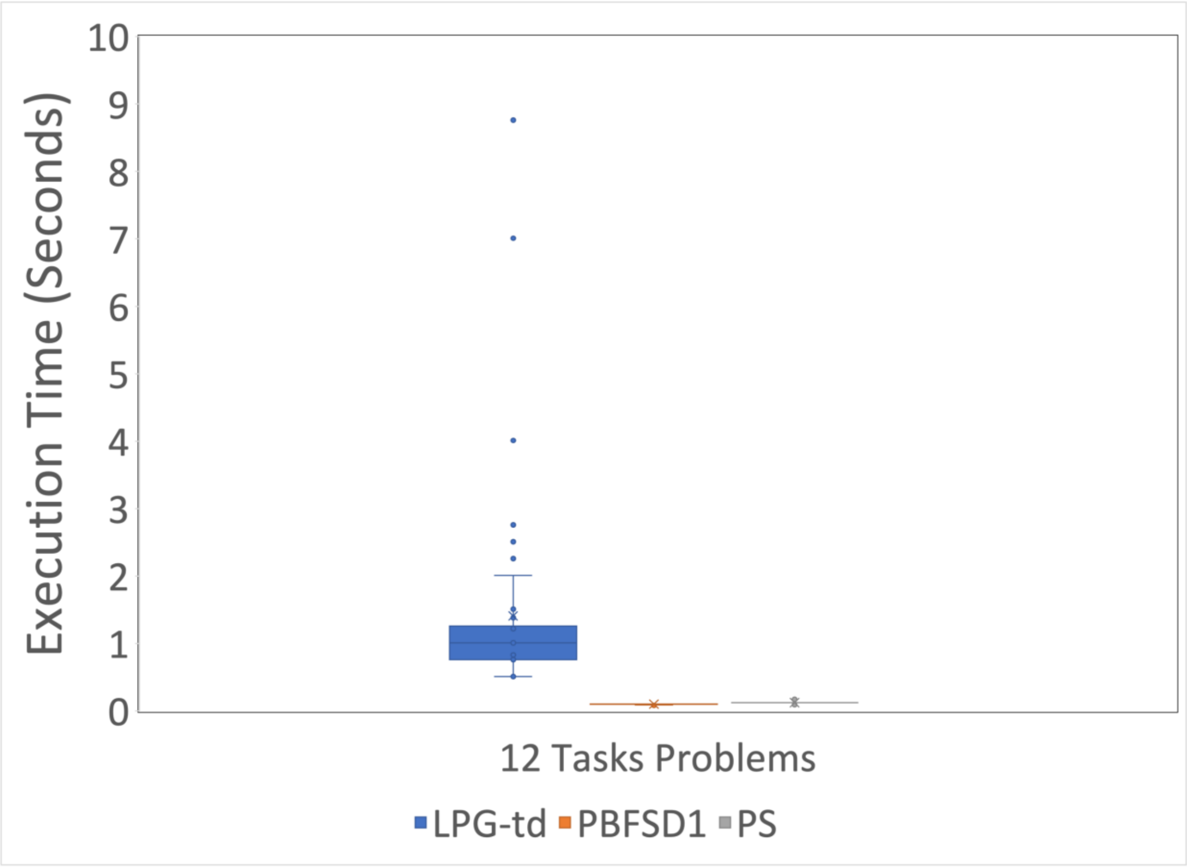


Fig. 5.10. LPG-td Execution time for problems with 12 tasks

5.2.2. Model-ExE problems

Time

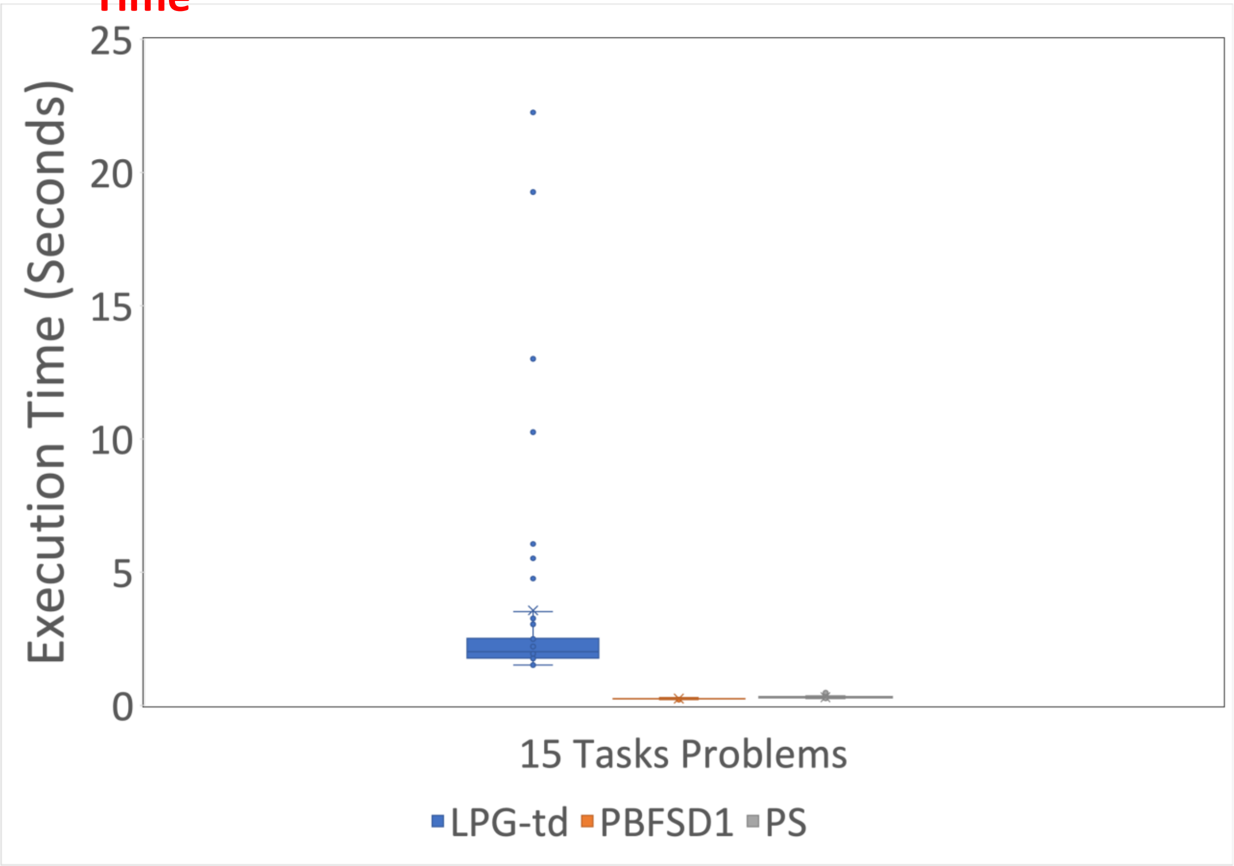


Fig. 5.11. Execution time for problems with 15 tasks

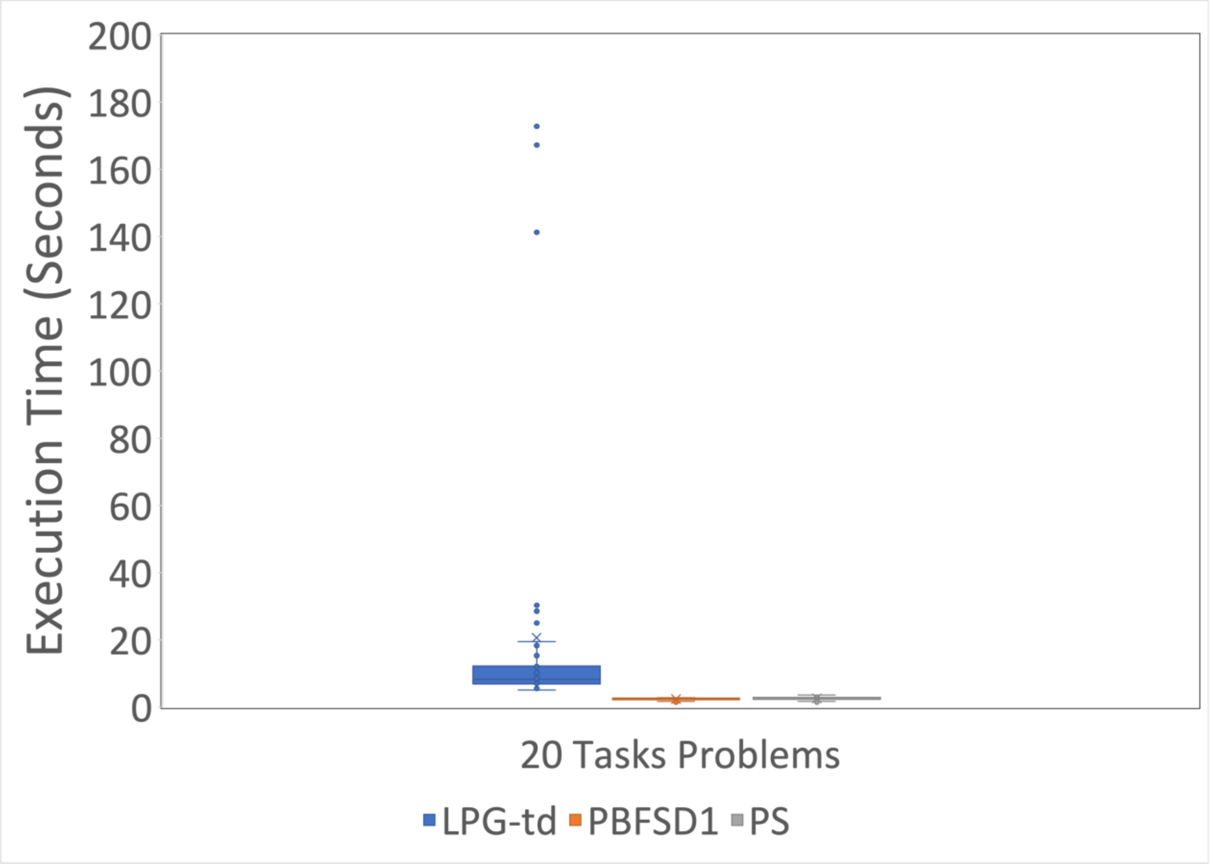


Fig. 5.12. LPG-td Execution time for problems with 20 tasks

5.2.2. Model-ExE problems

Time

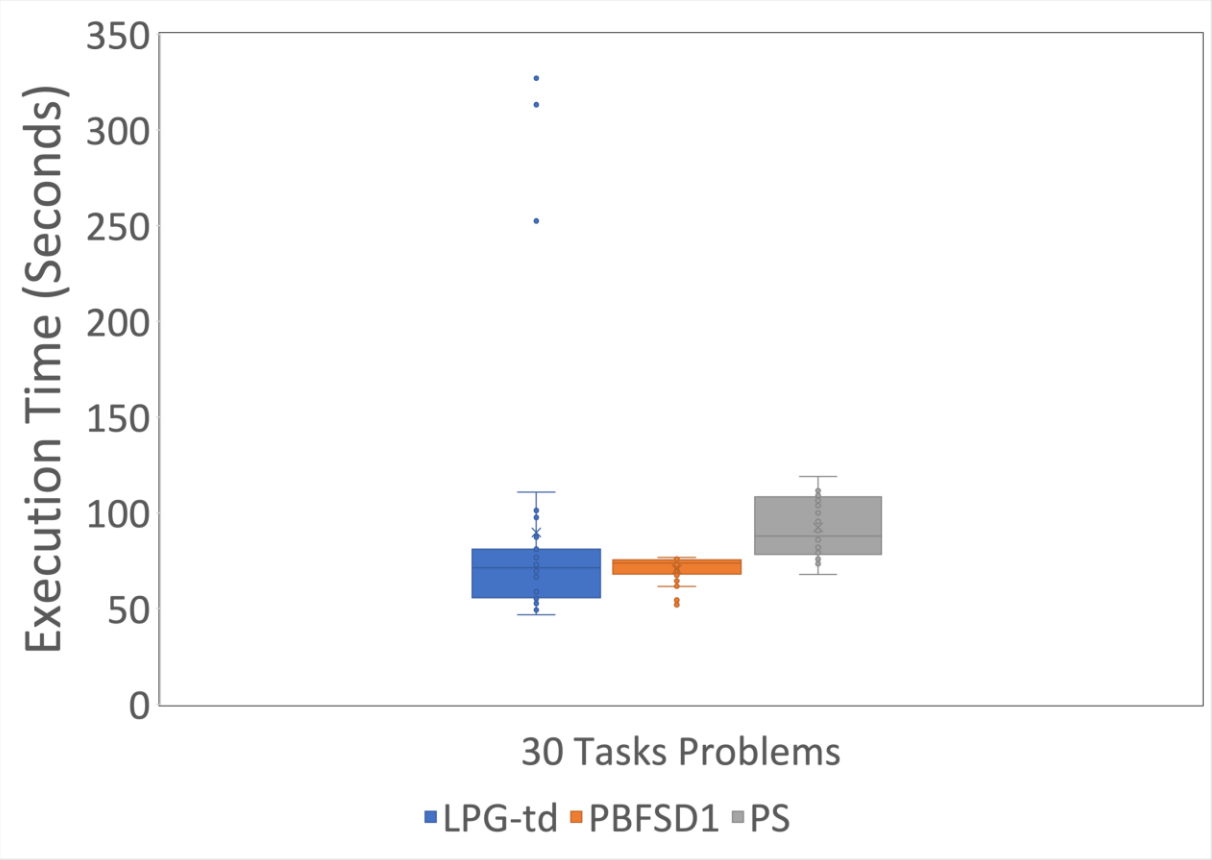


Fig. 5.13. LPG-td Execution time for problems with 20 tasks

5.2.2. Model-ExE problems

Quality of solutions

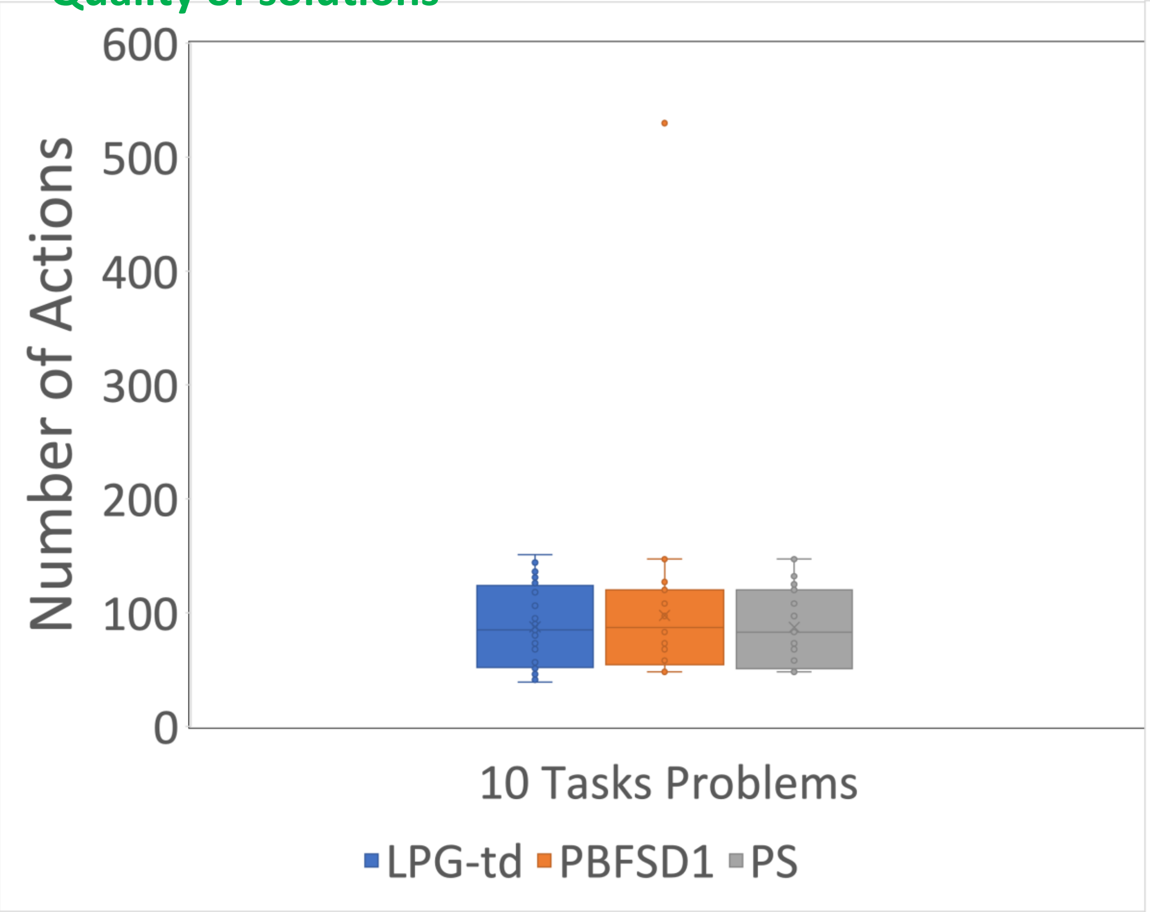


Fig. 5.14. Number of actions for problems with 10 tasks

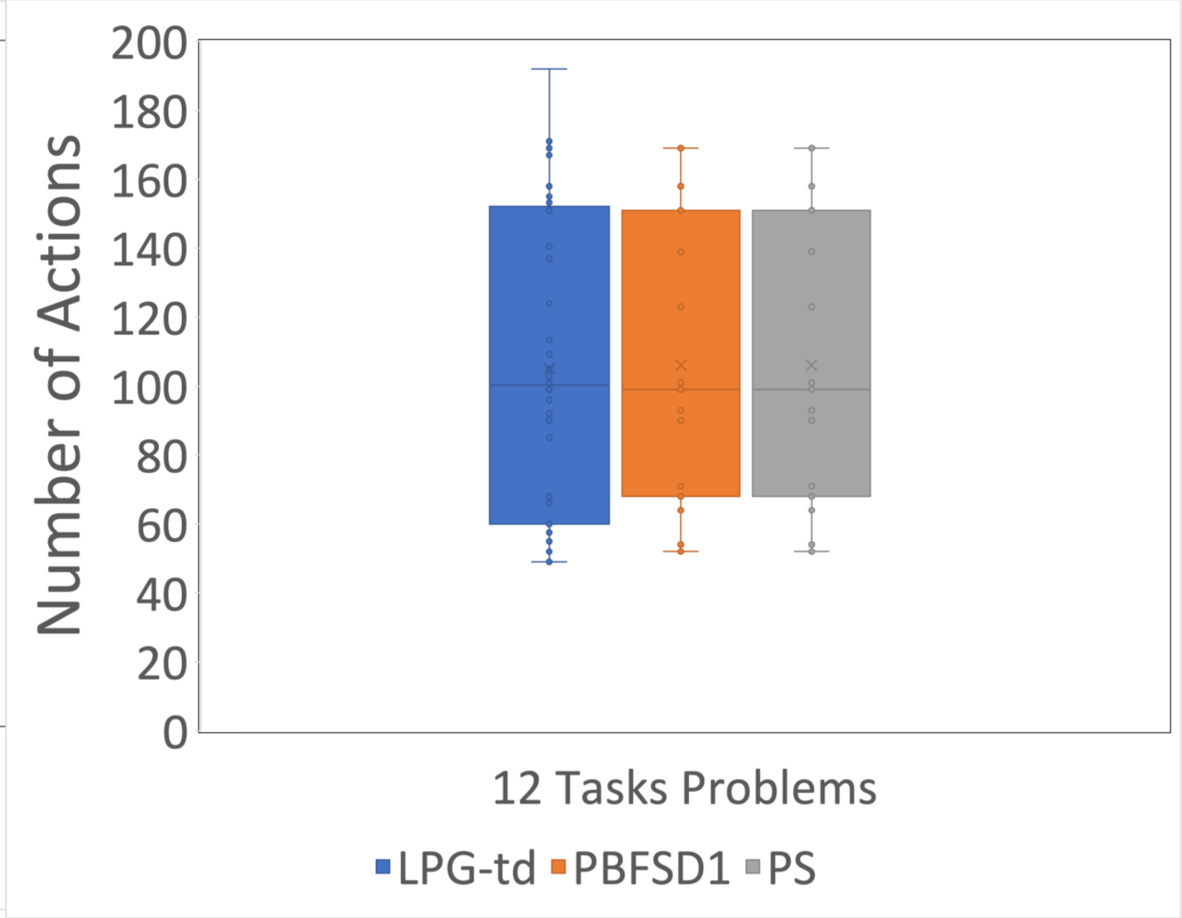


Fig. 5.15. Number of actions for problems with 12 tasks

5.2.2. Model-ExE problems

Quality of solutions

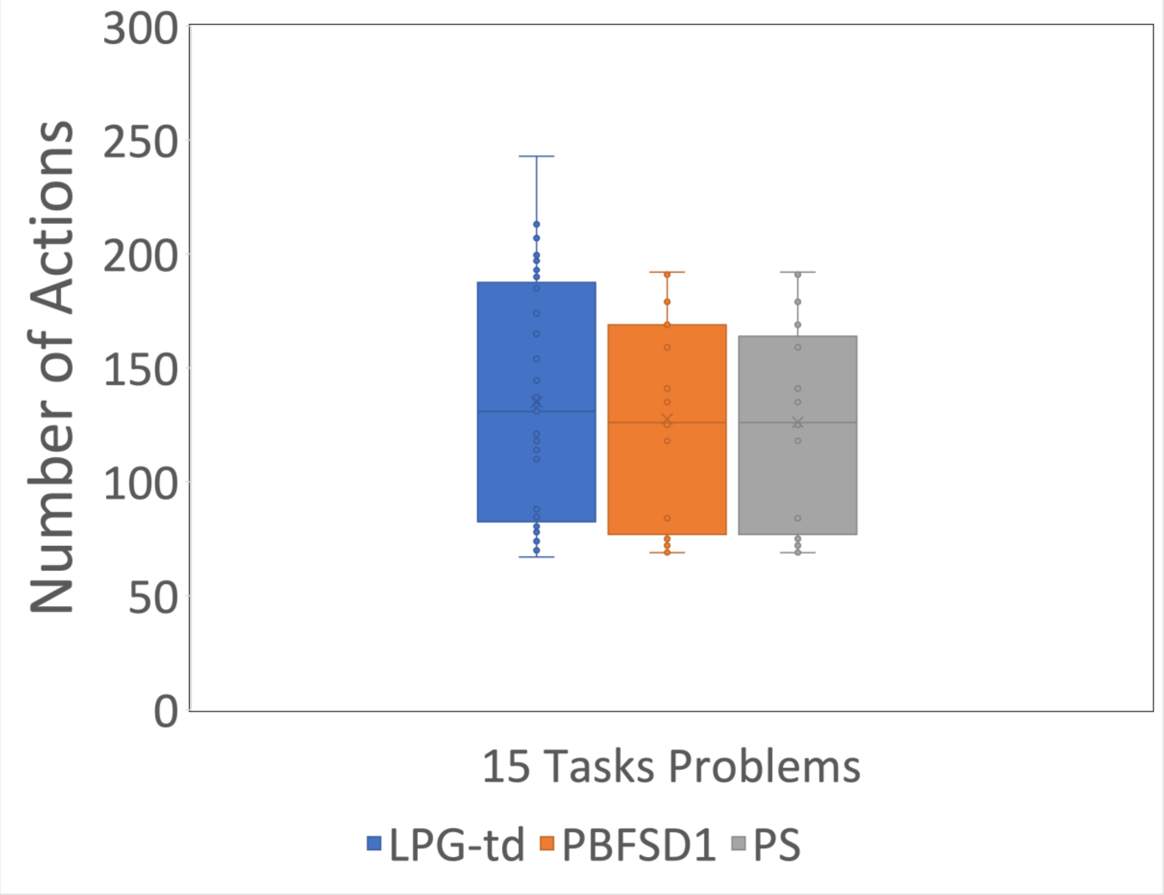


Fig. 5.16. Number of actions for problems with 15 tasks

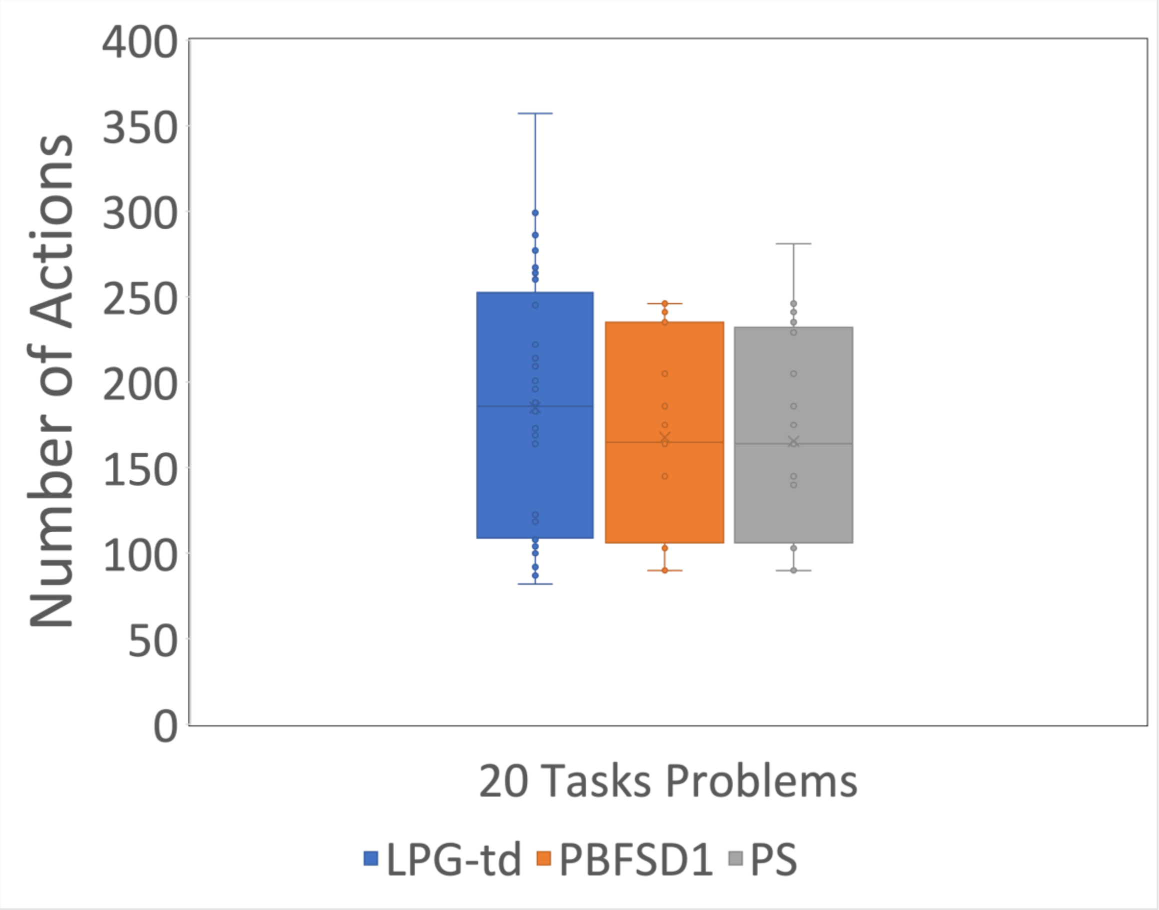


Fig. 5.17. Number of actions for problems with 20 tasks

5.2.2. Model-ExE problems

Quality of solutions

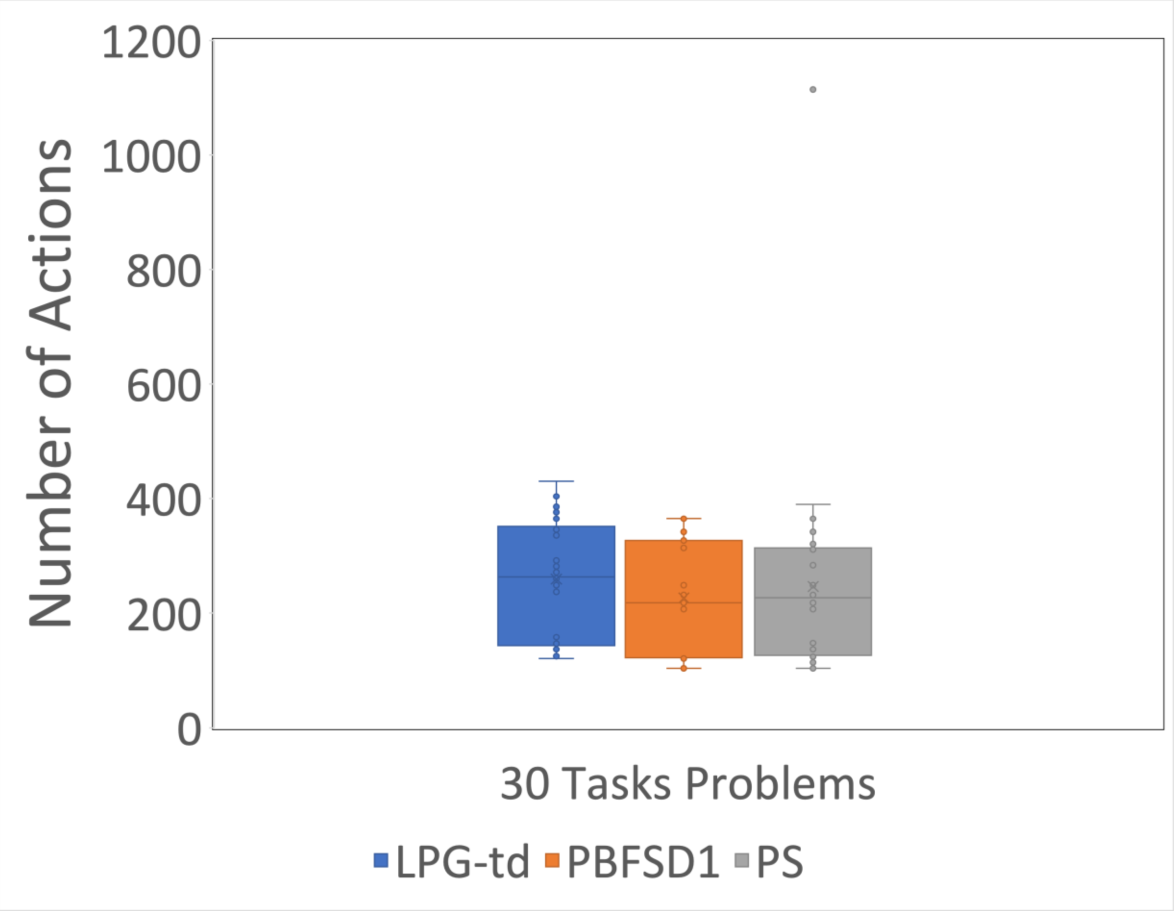


Fig. 5.18. Number of actions for problems with 30 tasks

5.2.2. Model-ExE problems

- The algorithms solved all instances of the evaluation set.
- The parallel versions were more efficient than LPG-td, returning globally better quality solutions.
- Solutions returned by PBFSD1 are 6% shorter than those of LPG-td, while PS solutions are also 5% better.
- LPG-td returns a solution on 23.21 s on average.
- PBFSD1 takes 14.65 s and *PS* 19.08 s.

5.2.3. Public transportation

Time

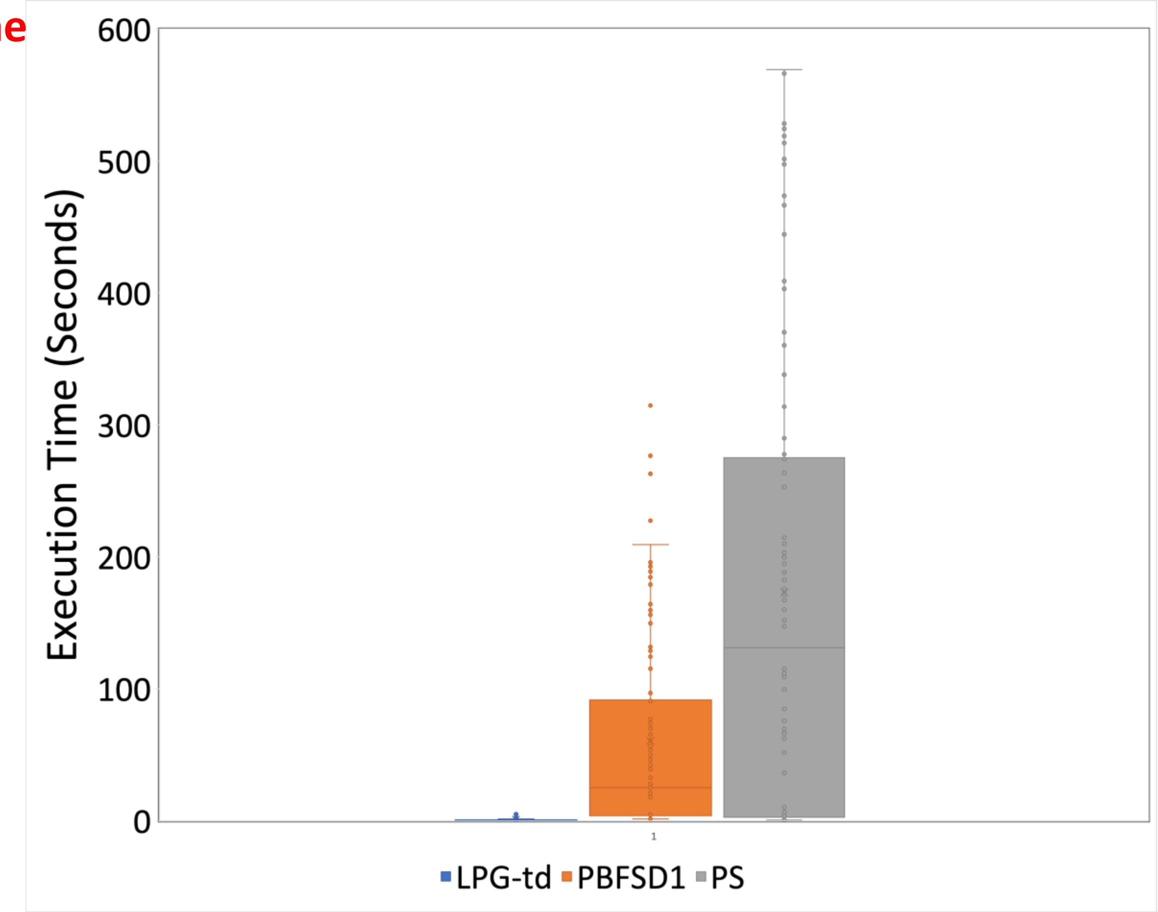


Fig. 5.19. Execution time

5.2.3. Public transportation

Quality of solutions

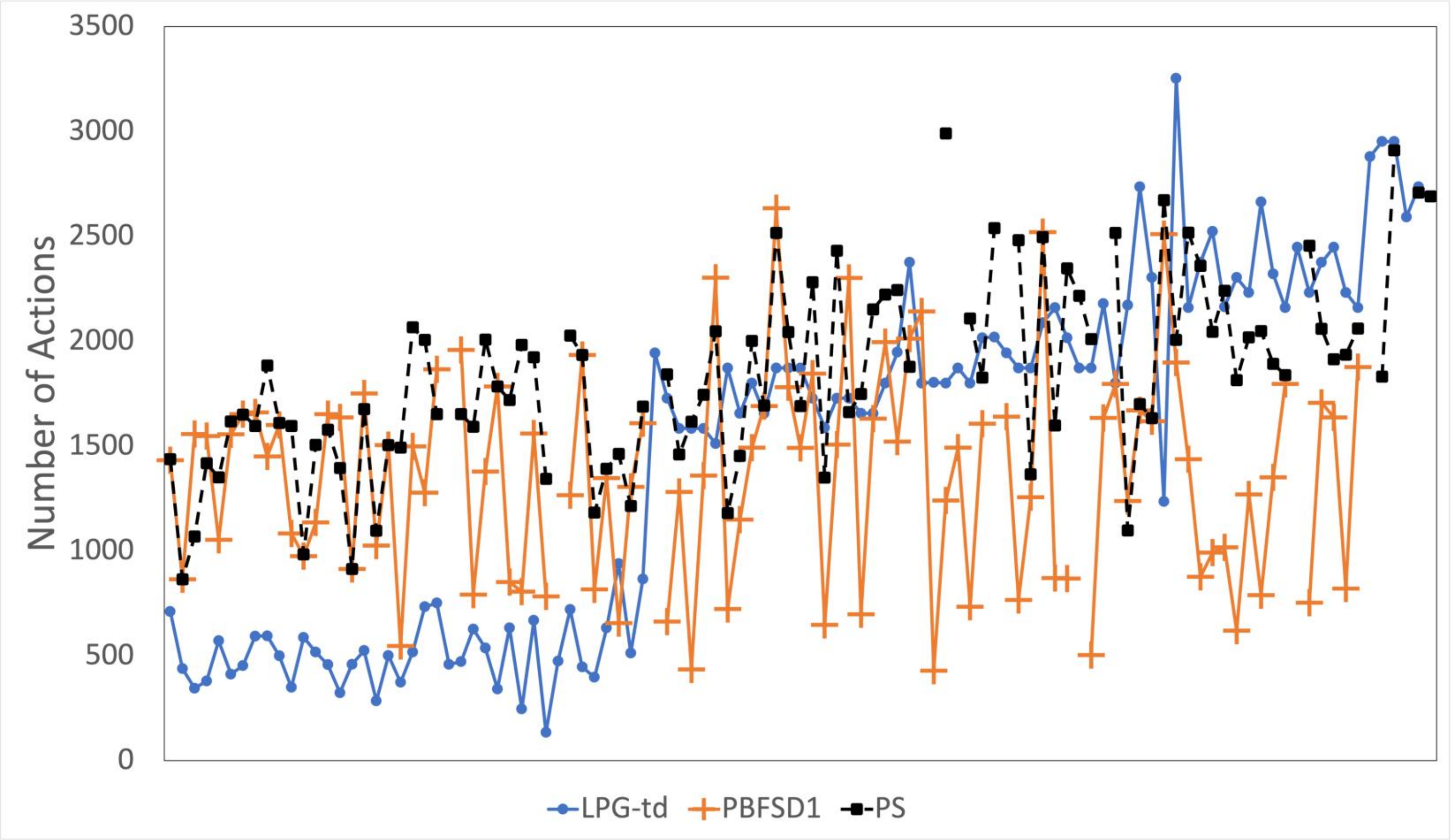


Fig. 5.20. Number of actions

5.2.3. Public transportation

- LPG-td solves 99% of the 105 problem instances.
- PS solves 89% of instances.
- PBFSD1 solves 88%.

➢ PBFSD1 algorithm returns on average shorter plans.

➢ PBFSD1 is 65% faster than PS.

5.3. Domains of IPC

- 269 problem instances from eight different planning domains.

- OpenStacks;
- Satellite;
- Pipes;
- PSR;
- Airport;
- Rovers;
- Promela;
- Pathway.

5.3.1. Openstacks

- OpenStacks domain is based on the simultaneous min–max open stack combinatorial optimization problem.
- A manufacturer has several orders, each for a combination of different products and can only make one product at a time.

5.3.1. Openstacks

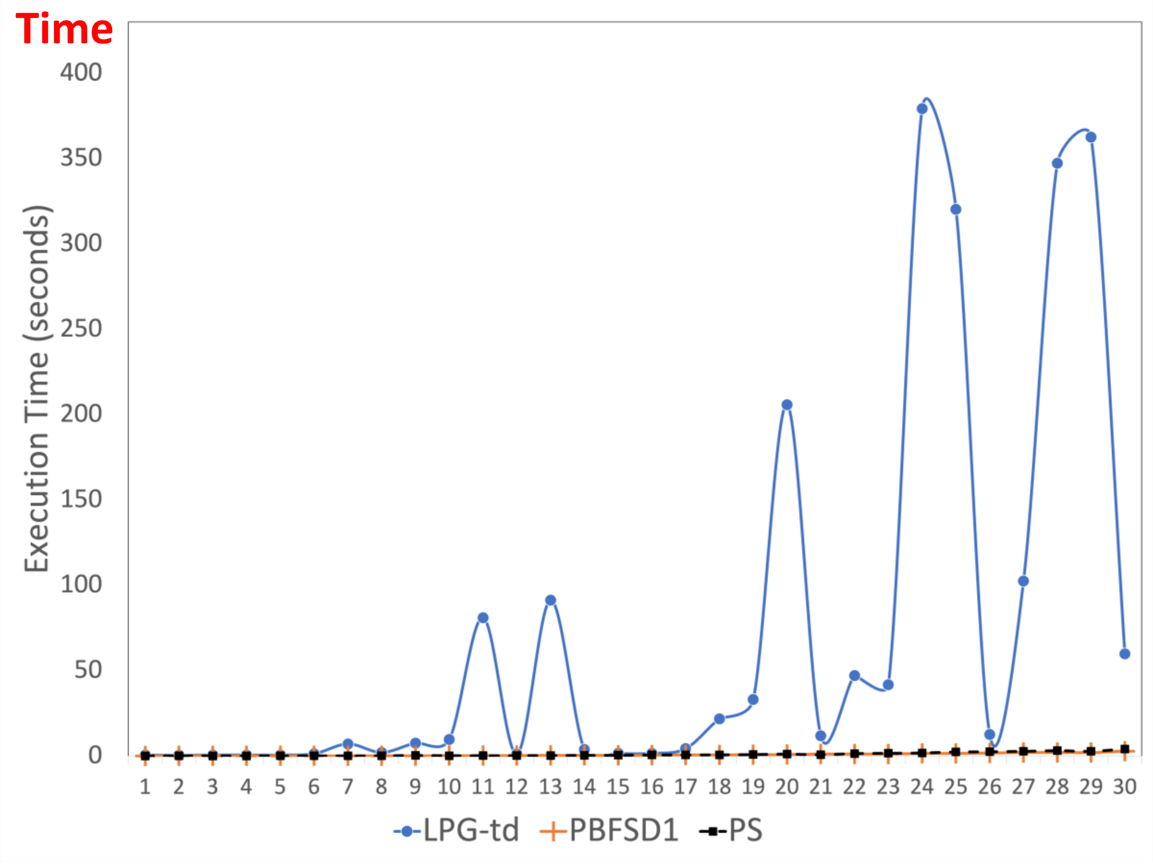


Fig. 5.21. Execution time

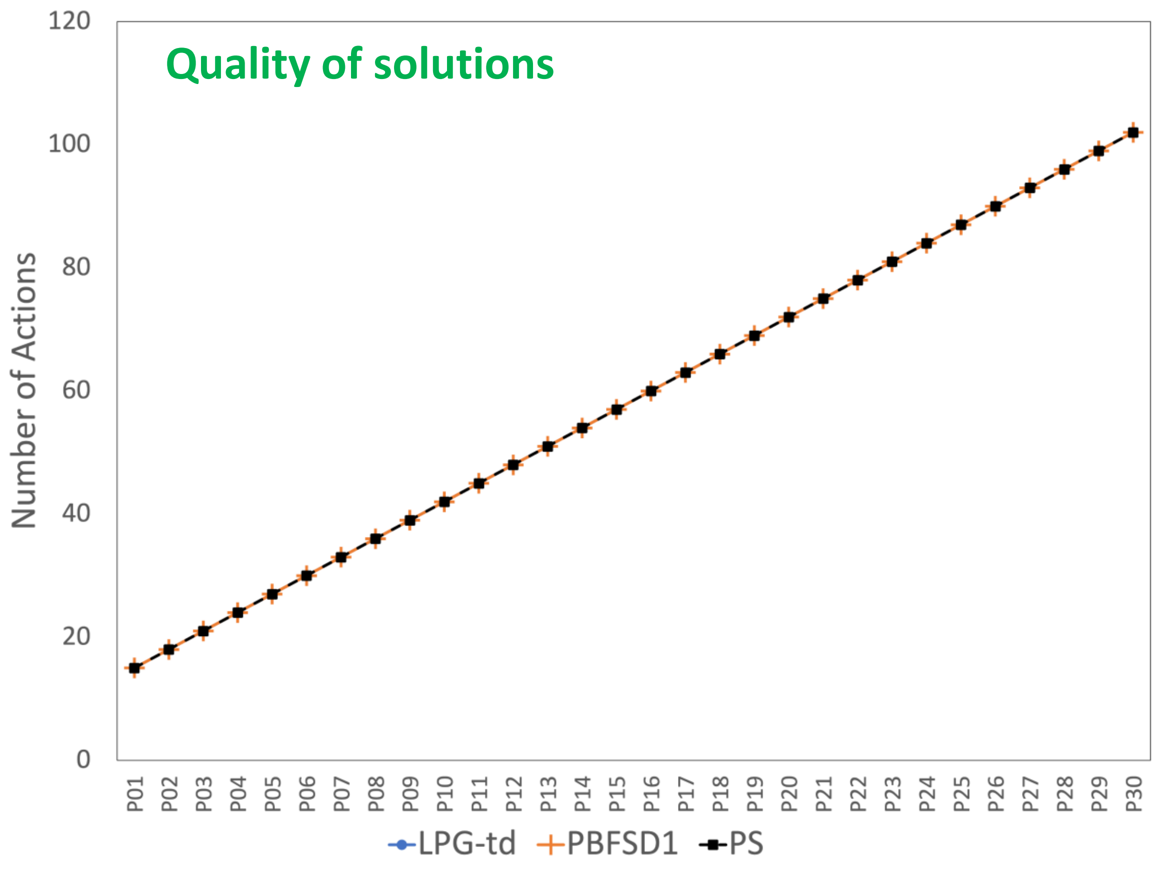


Fig. 5.22. Number of actions

Parallel algorithms perform consistently better than LPG-td.
They are sometimes 200 times faster without losing solution quality.

5.3.2. Satellite

- This domain considers a set of satellites equipped with different devices that operate under various modes.
- The objective is to acquire images. Satellites divide the observation tasks considering the capabilities of their instruments.

5.3.2. Satellite

Time

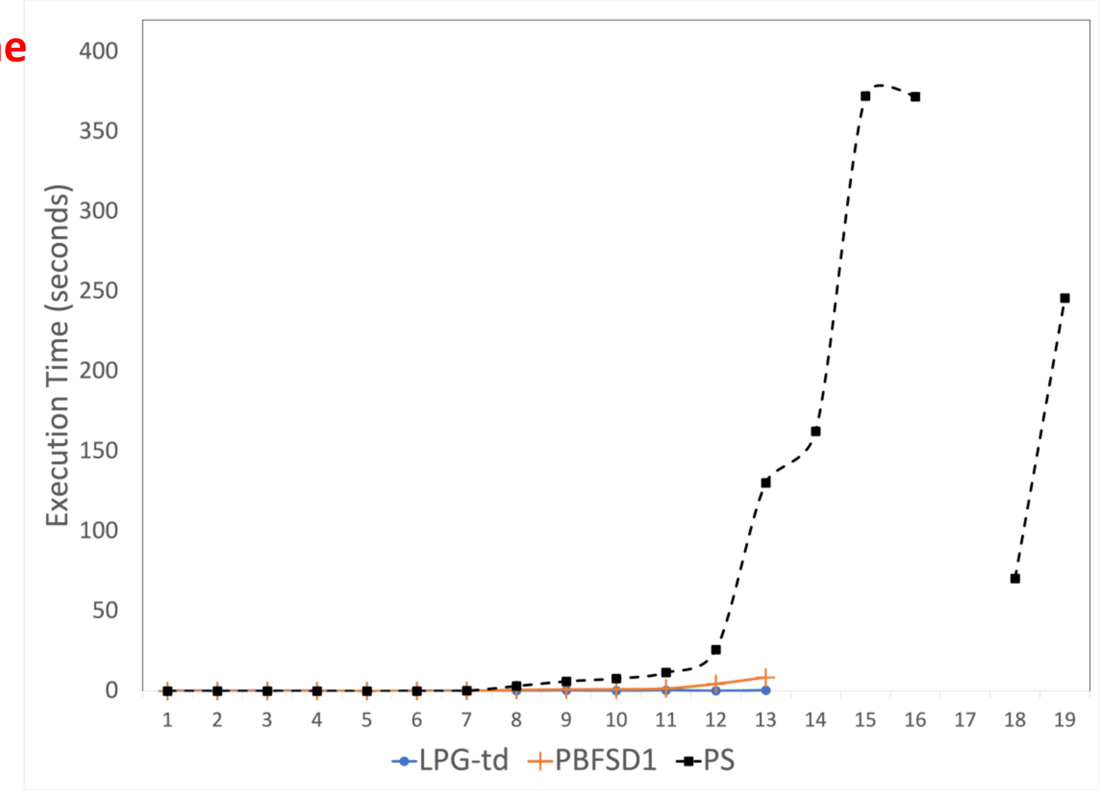


Fig. 5.23. Execution time

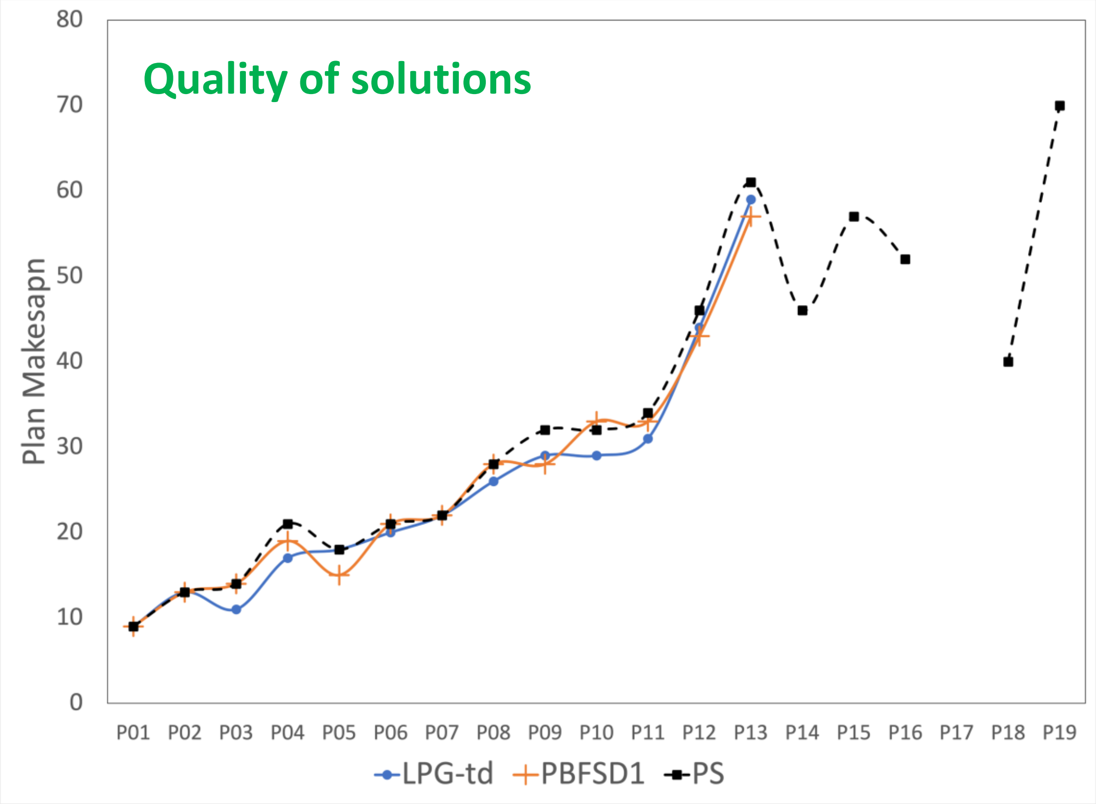


Fig. 5.24. Plan makespan

PBFSD1 solves the same first 13 problems as LPG-td and is competitive in terms of number of actions and execution time.
PS can scale up to more problems.

5.3.3. Pipes world

- Planners control the flow of oil derivatives through a pipeline network.

➢ Obeying various constraints such as product compatibility, tankage restrictions, and (in the most complex domain version) goal deadlines.

5.3.3. Pipes world

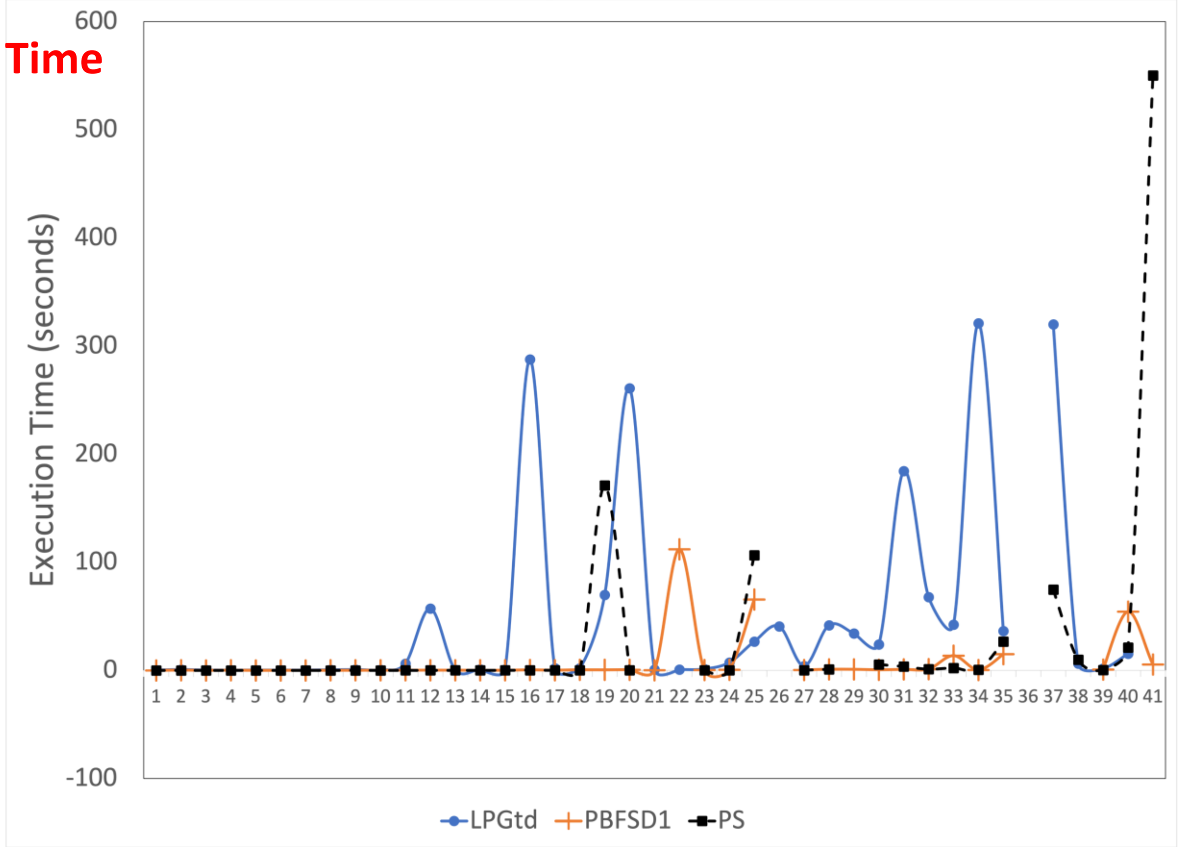


Fig. 5.25. Execution time

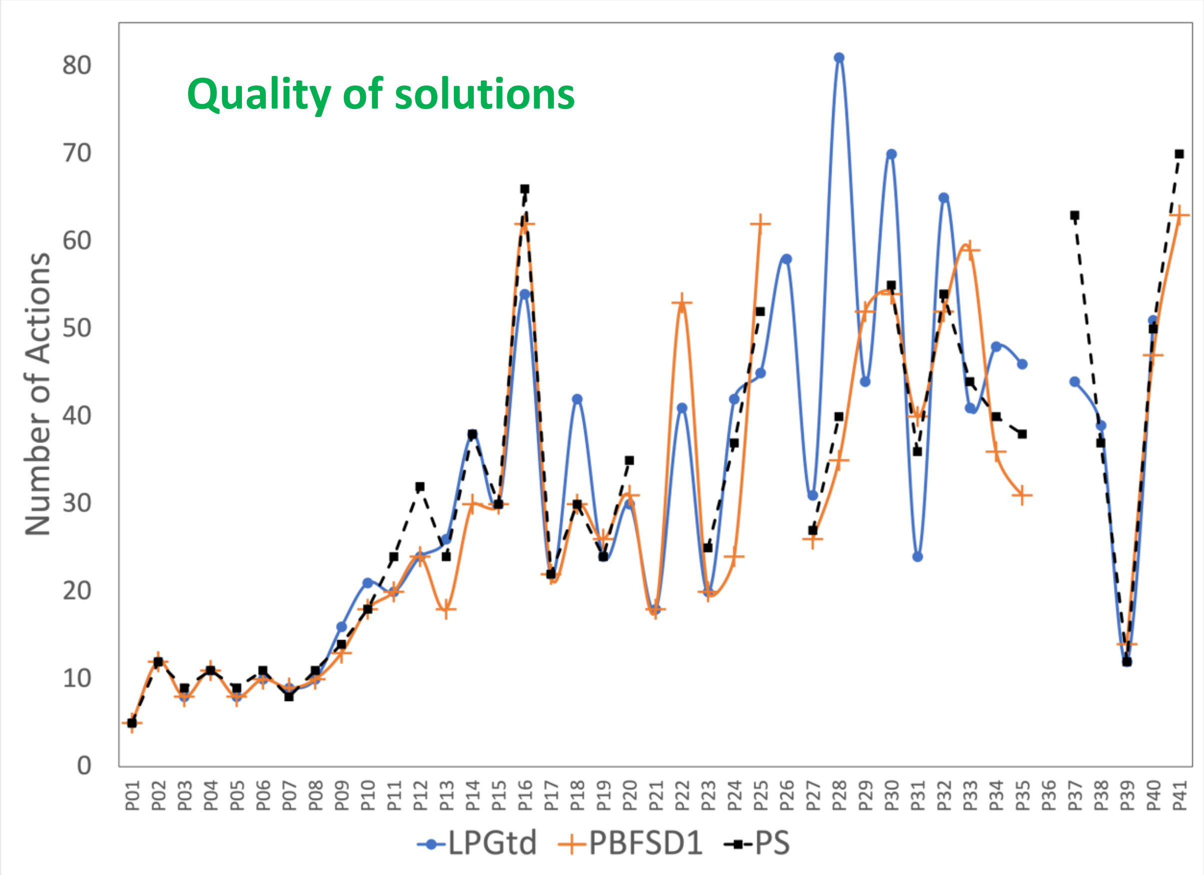


Fig. 5.26. Number of actions

LPG-td solves more problems than PBFSD1 and PS, respectively.

However, parallel algorithms perform generally better than LPG-td:

PBFSD1 returns better quality solutions at a fraction of the time taken by the other approaches.

5.3.4. PSR

- The PSR domain considers resupplying lines in a faulty electricity network.
- A transitive closure over the network connections determines its flow of electricity.
- This flow is subject to the states of the electric supply devices.

5.3.4. PSR

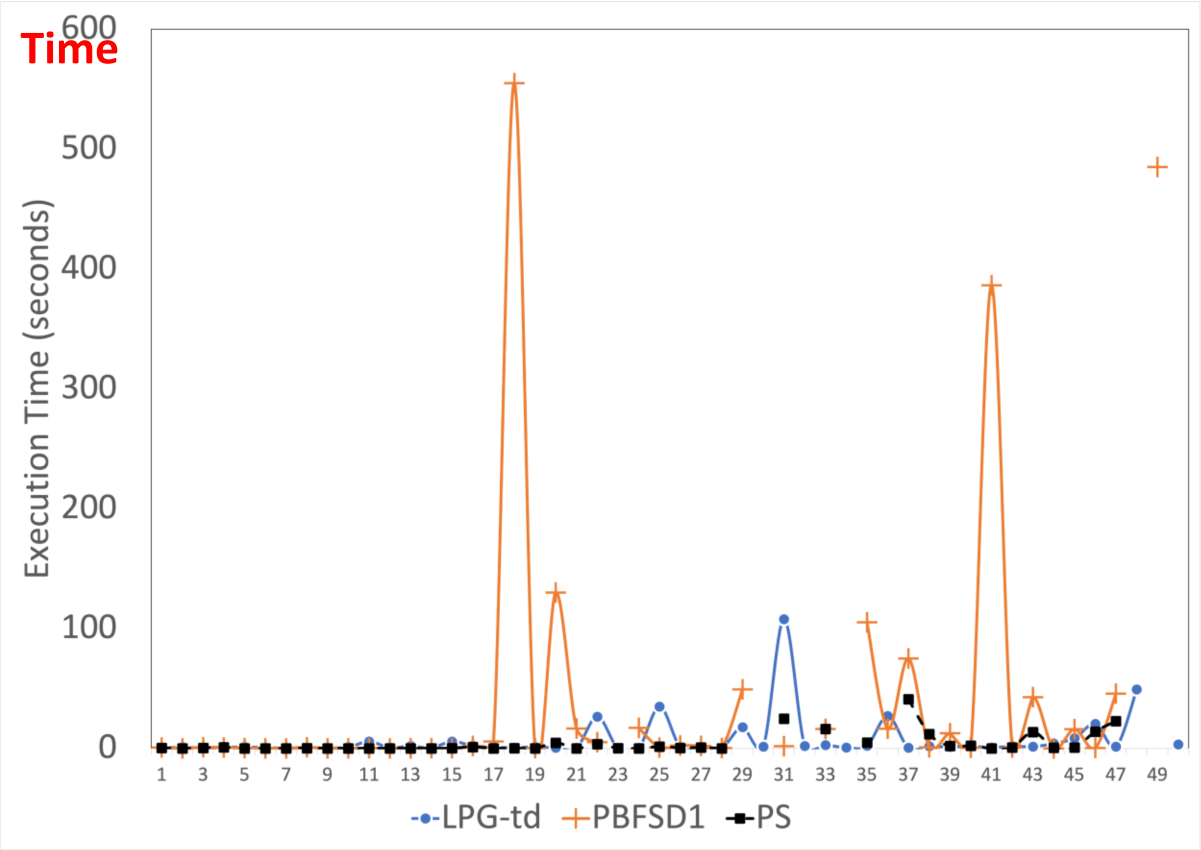


Fig. 5.27. Execution time

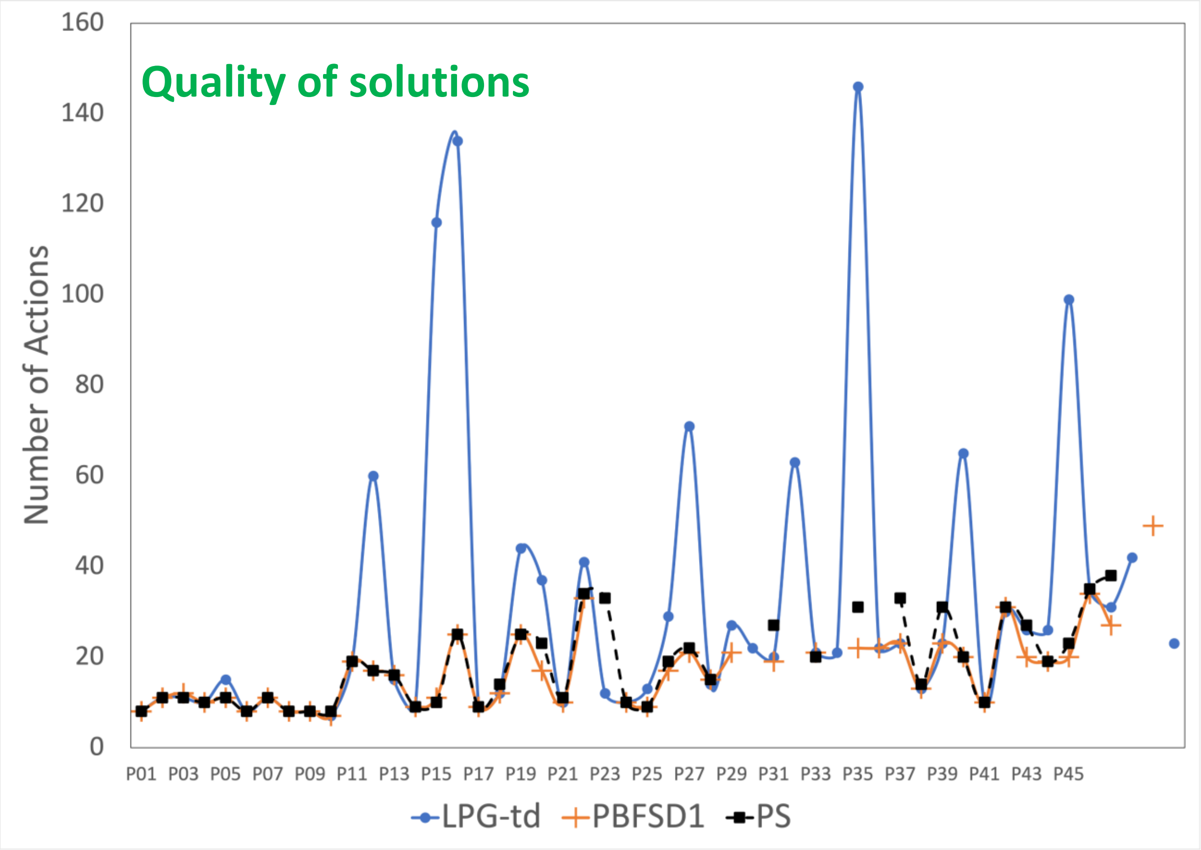


Fig. 5.28. Number of actions

Parallel algorithms generally return better quality solutions. However, they solve 12% fewer problems than LPG-td.

5.3.5. Airport

- One of the most complex domains for the parallel methods is Airport.
- This domain considers the problem of planning ground traffic operations at an airport.
- The airport scenarios illustrate traffic situations arising during simulation runs in the airport simulation tool Astras.
- The largest instances in the test suites are realistic encodings of the Munich airport.

5.3.5. Airport

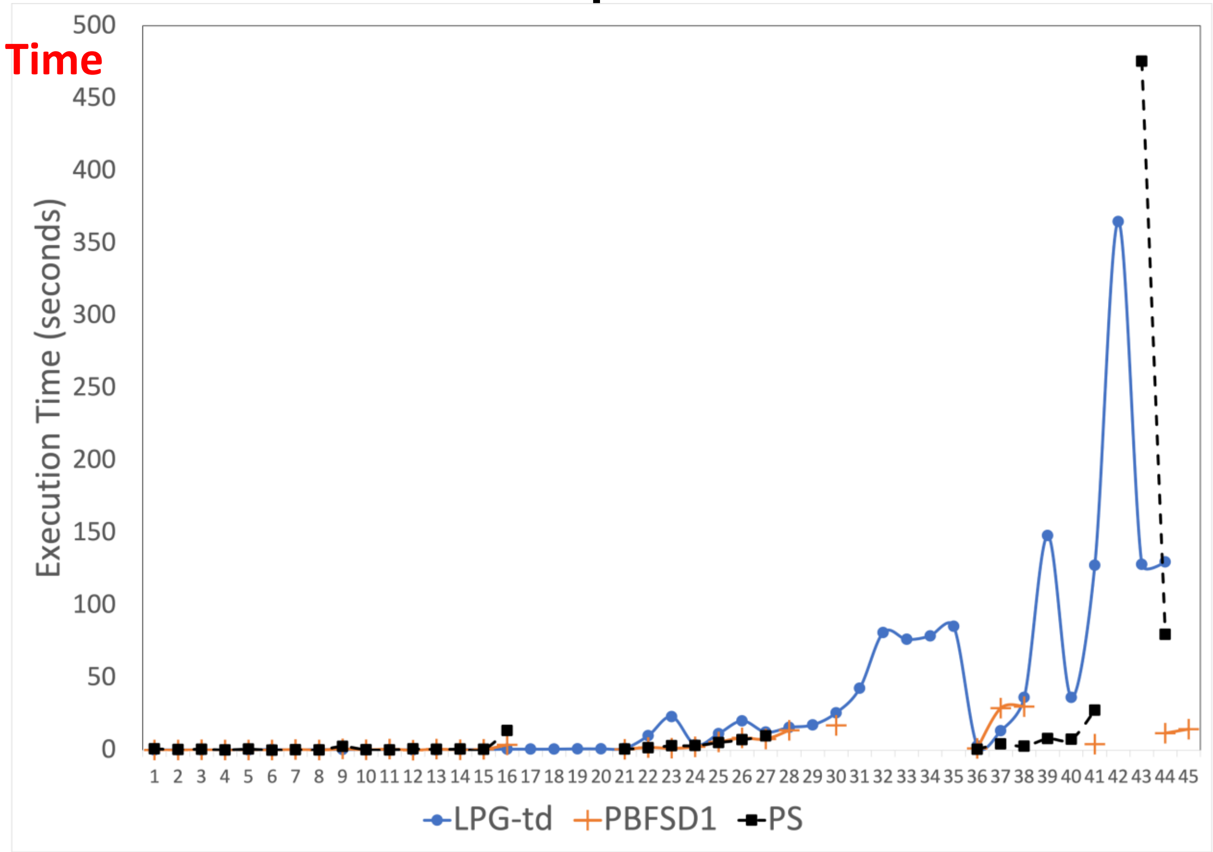


Fig. 5.29. Execution time

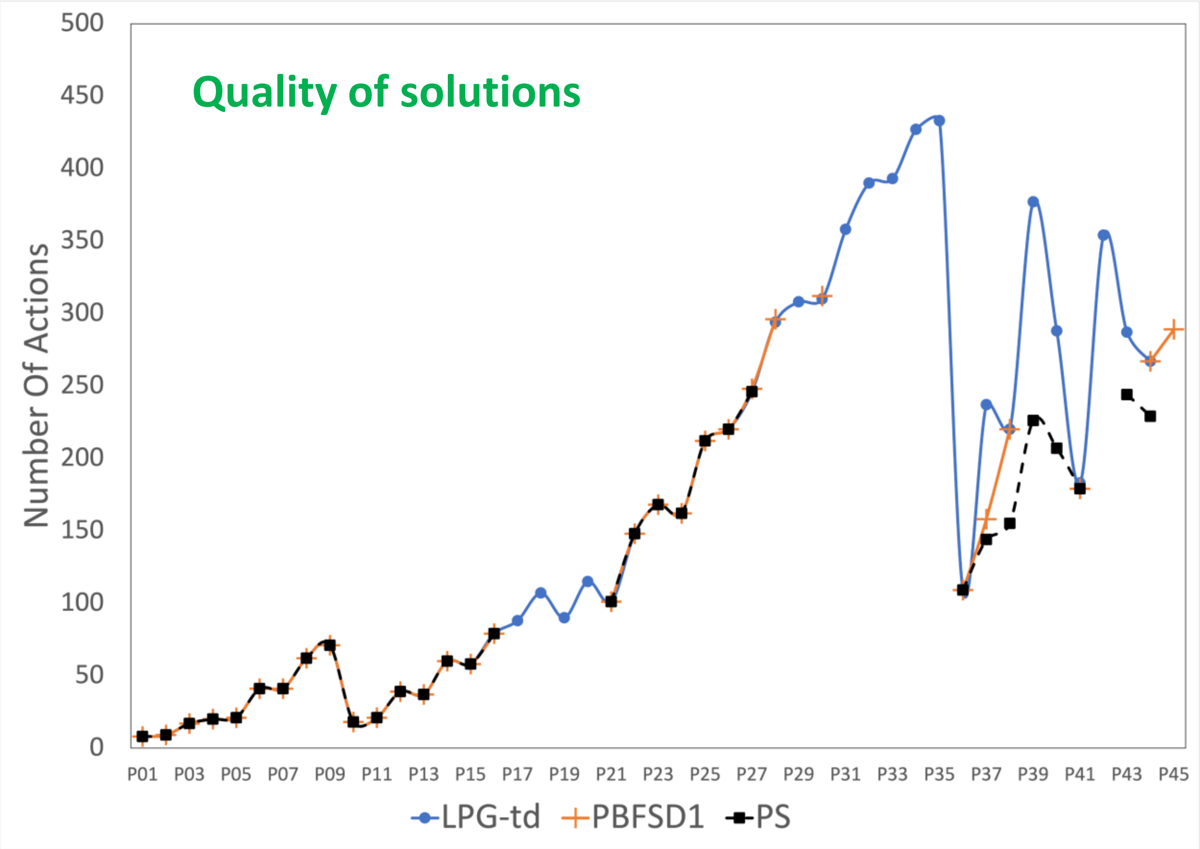


Fig. 5.30. Number of actions

LPG-td solved 98% of the scenarios.

PBFSD1 and PS found plans for 69% of the scenarios.

For difficult instances, PS tends to give better solutions in terms of time and number of actions than LPG-td.

5.3.6. Rover

- The Rovers domain models a collection of rovers that must navigate a planet’s surface.
- Rovers must collect samples and communicate data about them to the lander.

5.3.6. Rover

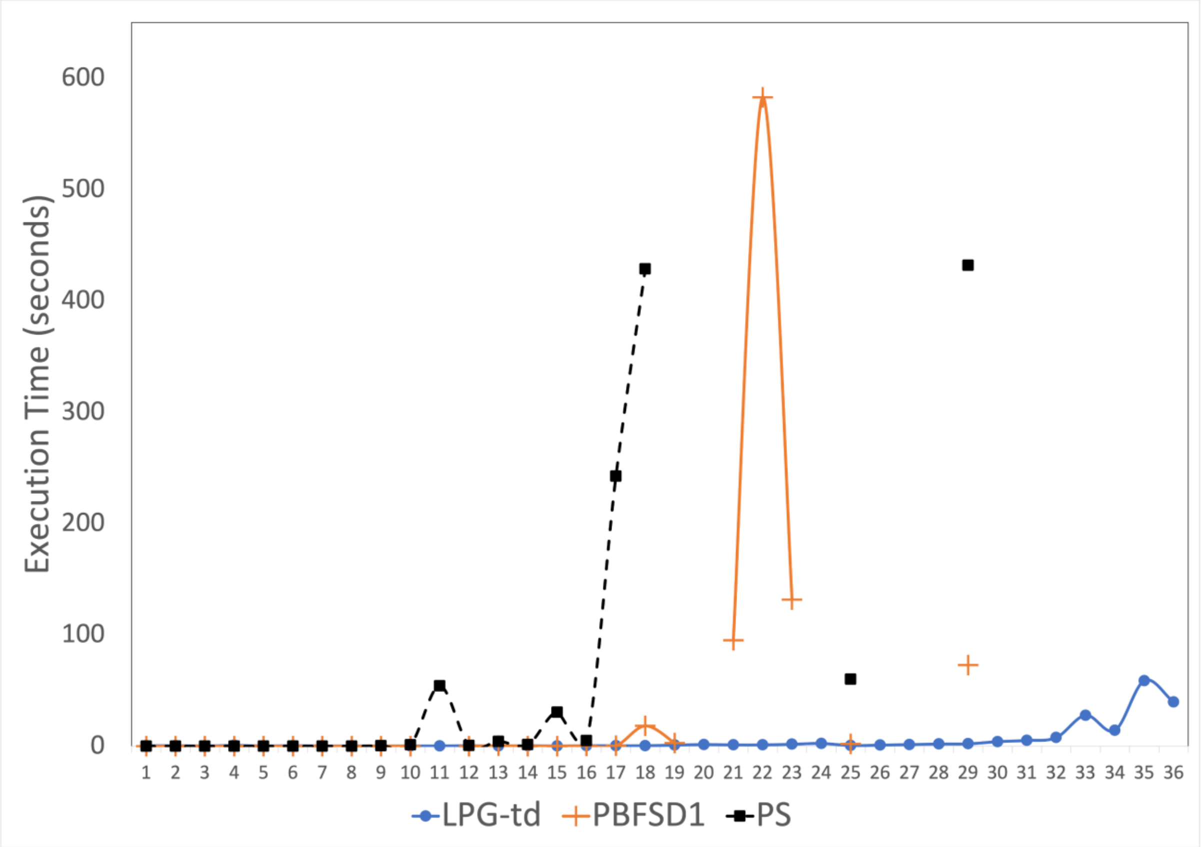


Fig. 5.31. Execution time

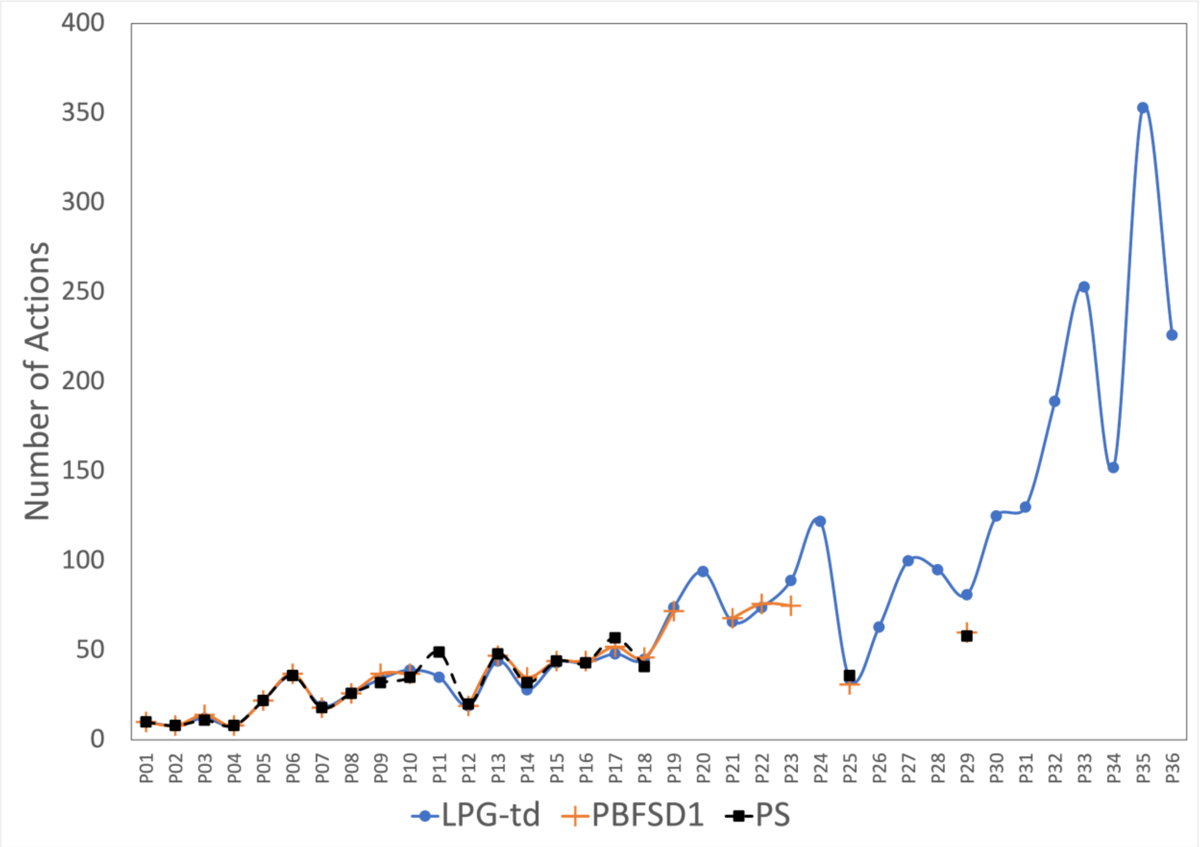


Fig. 5.32. Number of actions

LPG-td solves all problems; PBFSD1 solves 63% and PS solves 55%.

Time performance for LPG-td is significantly superior.

Parallel algorithms return solutions whose quality is equivalent to that provided by LPG-td.

5.3.7. Promela

- Promela models deadlocks in communication protocols.
- Processes that emulate finite-state transition diagrams model the deadlocks.
- The communication protocols used in Promela consider the dining philosophers problem and the telegraph routing problem.

5.3.7. Promela

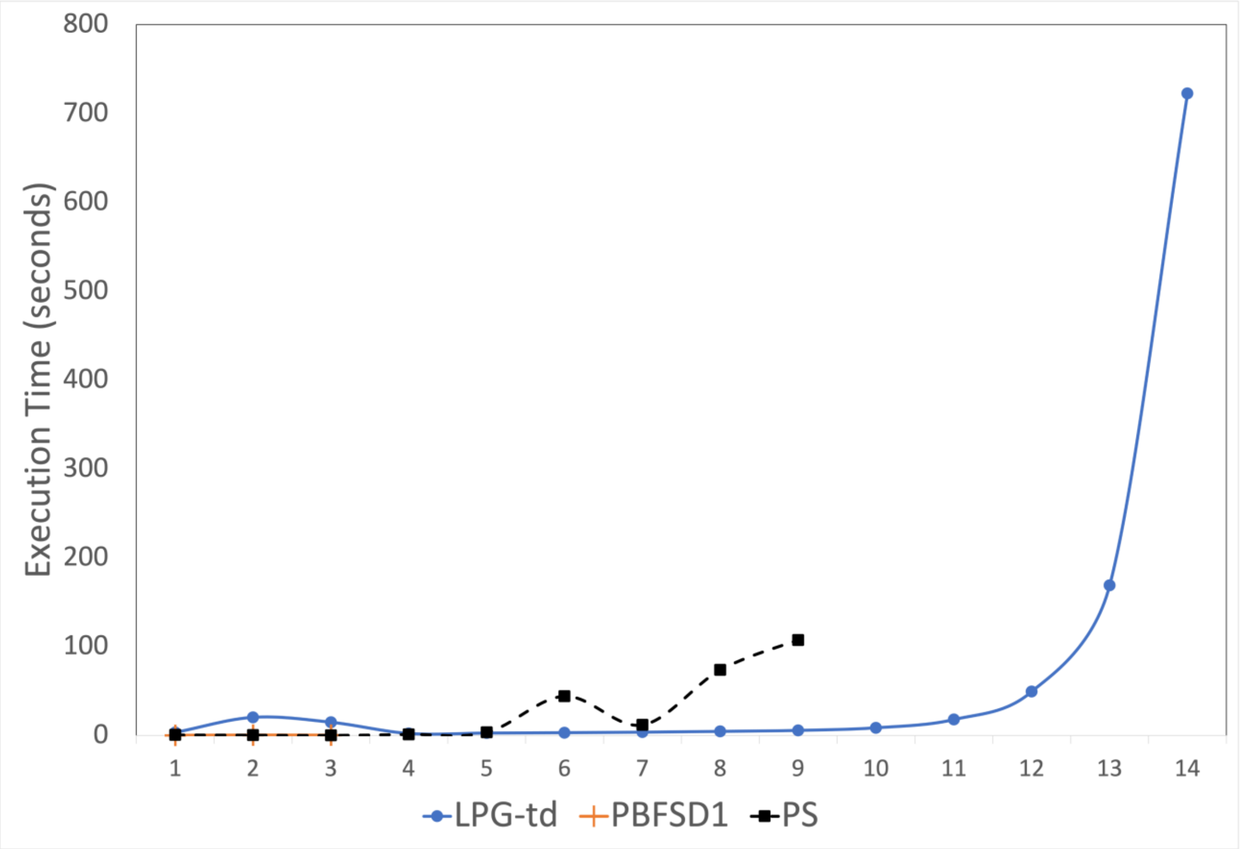


Fig. 5.33. Execution time

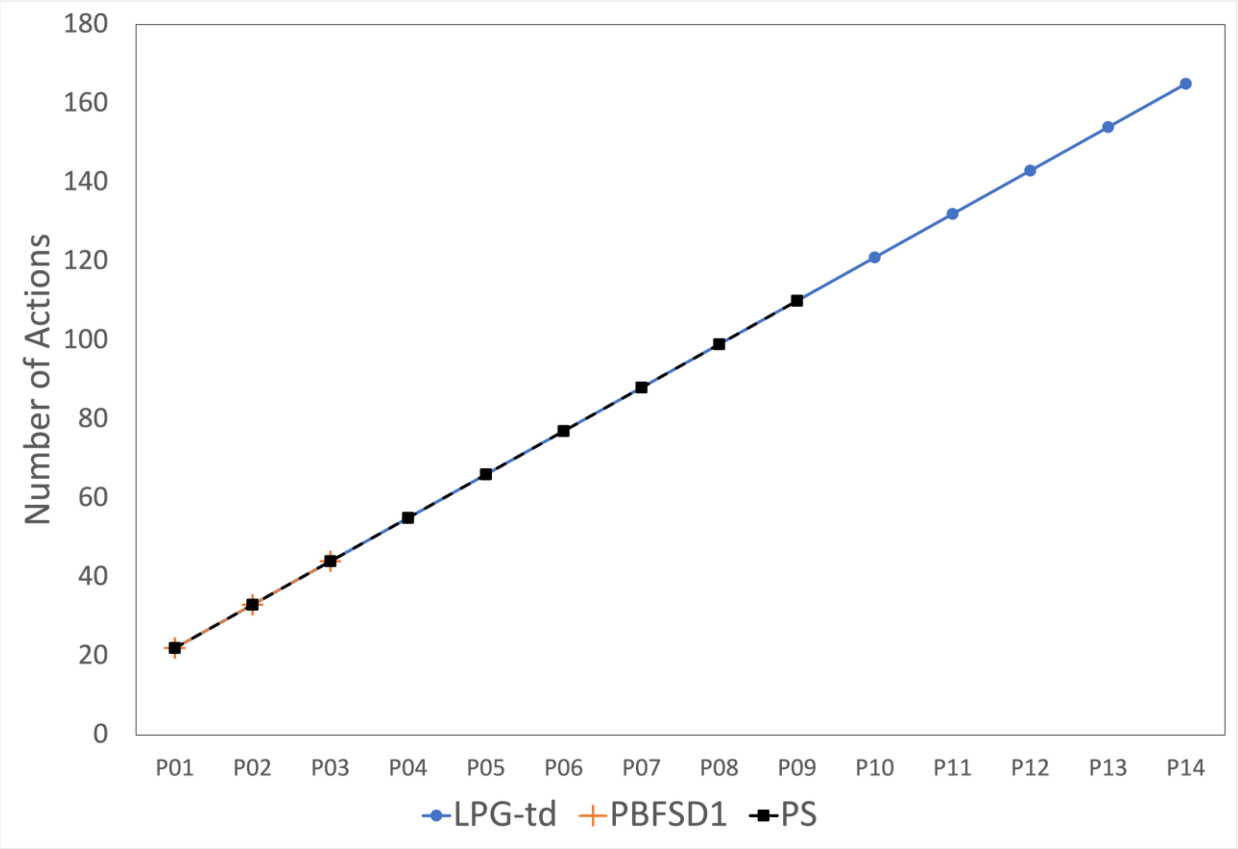


Fig. 5.34. Number of actions

LPG-td solves 14 scenarios; PS solves 9 (64%); PBFSD1 solves only 3 (21%).
All the algorithms return the same quality solutions in terms of number of actions.

5.3.8. Pathway

- The Pathway domain models biochemical pathways, i.e., chemical reactions in a biological organism that explain cell behavior.
- The goal consists of synthesizing specific substances in the pathway.
- This model is the most complex domain evaluated for the parallel algorithms.

5.3.8. Pathway

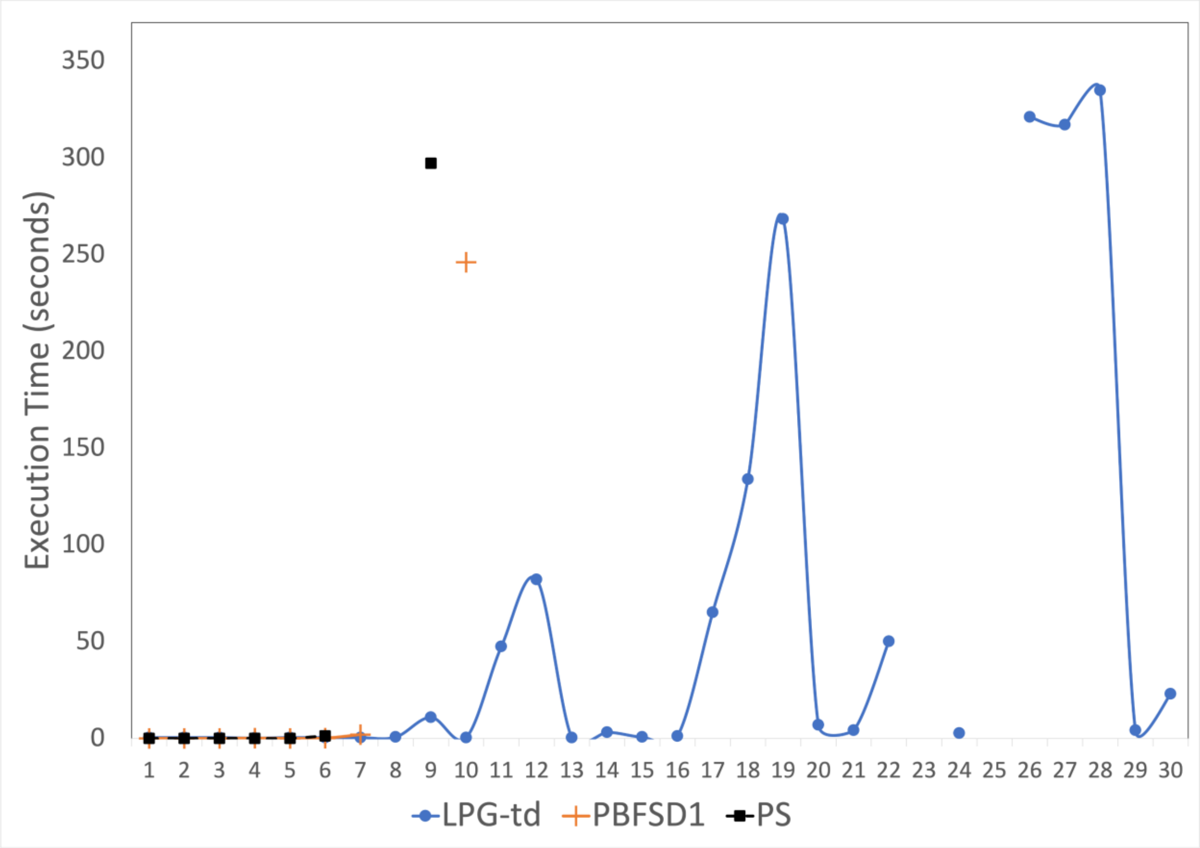


Fig. 5.35. Execution time

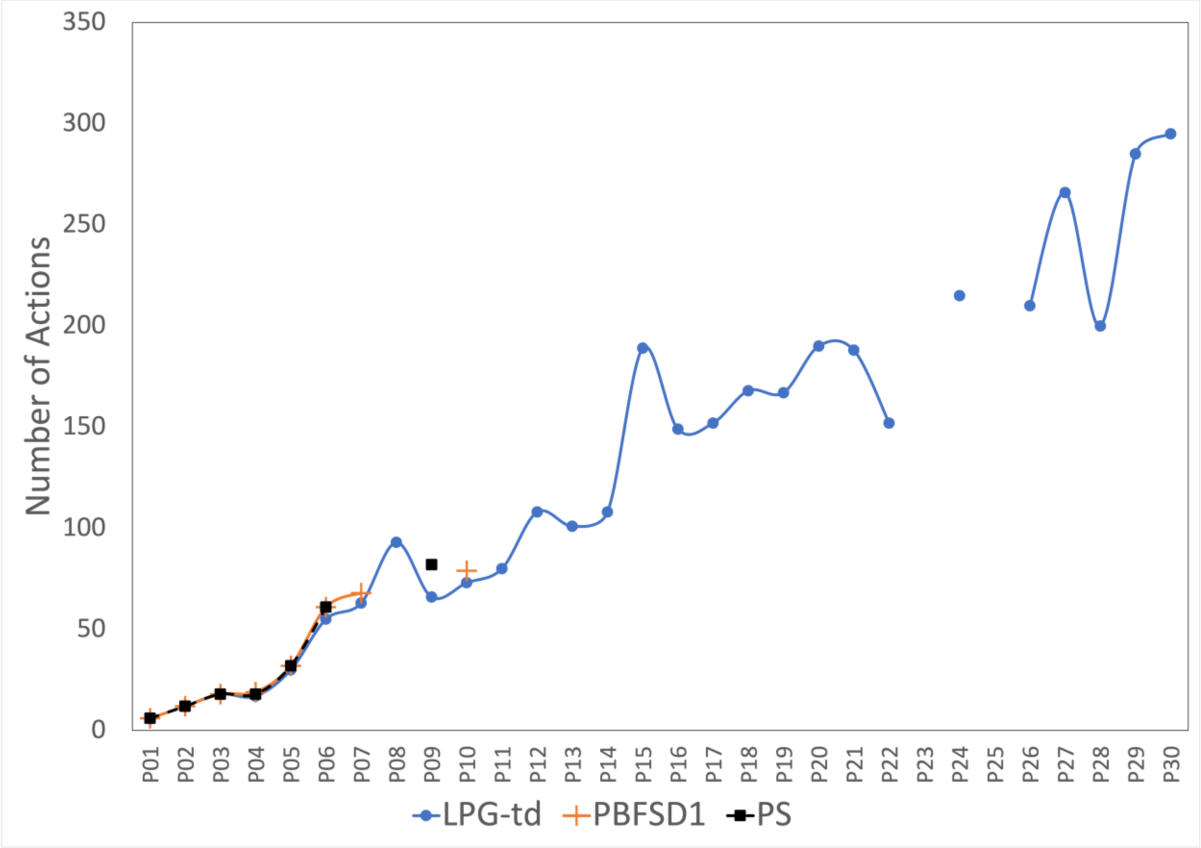


Fig. 5.36. Number of actions

LPG-td finds a solution in 93% of the scenarios.

PBFSD1 solves only 26% of the problems.

PS returns solutions in 23% of them.

5.4. Solutions for 4,8,16 and 32 threads

PBFSD1	Sequential		4 threads		8 threads		16 threads		32 threads	
	time	actions	time	actions	time	actions	time	actions	time	actions
P24	**0.04**	38	0.10	45	0.51	38	0.054	38	0.54	24
P27	–	–	0.16	52	**0.04**	28	–	–	0.40	26
P31	–	–	–	–	130.29	65	–	–	**0.92**	40
P34	0.18	56	0.07	42	**0.06**	38	0.09	42	0.45	36

PS	Sequential		4 threads		8 threads		16 threads		32 threads	
	time	actions	time	actions	time	actions	time	actions	time	actions
P24	0.40	36	**0.05**	36	0.11	32	0.34	41	0.12	37
P27	0.17	28	0.31	26	0.61	27	**0.15**	26	0.26	27
P31	16.09	43	2.24	48	**0.67**	38	4.2	43	3.57	36
P34	0.43	44	1.73	52	**0.28**	44	0.94	42	0.58	40

Pipes World problems

5.4. Solutions for 4,8,16 and 32 threads

PBFSD1	Sequential		4 threads		8 threads		16 threads		32 threads	
	time	actions	time	actions	time	actions	time	actions	time	actions
P15	**0.016**	58	0.80	58	0.82	58	0.61	58	0.61	58
P16	110.42	83	40.88	83	17.46	83	7.24	83	**3.80**	79

PS	Sequential		4 threads		8 threads		16 threads		32 threads	
	time	actions	time	actions	time	actions	time	actions	time	actions
P39	–	–	9.93	282	7.42	226	**4.74**	226	8.02	226
P40	21.06	265	–	–	–	–	**5.92**	207	7.55	207

Airport problems

5.4. Solutions for 4,8,16 and 32 threads

PBFSD1	Sequential		4 threads		8 threads		16 threads		32 threads	
	time	actions	time	actions	time	actions	time	actions	time	actions
P144-1	**0.02**	1296	0.69	1033	0.88	1042	1.97	1033	6.47	547
P225-1	**0.05**	1800	4.19	1751	6.03	1753	10.41	1753	39.17	1609

Public transportation problems

Cluster with 6 nodes, 48 cores					One node, 32 cores	
Jinnai [2]	A*	FAZHDA*	OZHDA*	AHDA*	**PBFSD1**	**PS**
Pipes NT 10	147.79	9.3	8.19	7.98	0.12	**0.05**
Jinnai [3]	A*	FAZHDA*	GAZHDA*	GRAZHDA*		
Pipes NT 10	157.31	10.6	10.1	10	0.12	**0.05**
Jinnai [3]	A*	DAHDA	ZHDA*	GRAZHDA*		
Pipes NT 12	201.07	6.11	9.12	7.71	0.21	**0.07**
Pipes NT 15	323.59	16.56	15.33	12.85	0.11	**0.1**
Jinnai [3]	A*	FAZHDA*	GAZHDA*	GRAZHDA*		
Rover 6	1042.69	25.76	31.13	25.32	**0.04**	0.12
Cloud cluster with 128 virtual cores					**One node, 32 cores**	
Jinnai [3]	A*	FAZHDA*	GAZHDA*	GRAZHDA*	**PBFSD1**	**PS**
Pipes NT 16	—	106.28	108.28	120.64	0.46	**0.19**
Airport 18	—	**95.48**	128.22	102.34	—	—

6. Conclusions and future work

- **Benefits derived from parallelism and in particular multithreading on modern multi-core CPUs and shared memory architectures for solving planning problems.**
- Two original parallel algorithms based on best-first search.

➢ Asynchronous updating of an ordered global list of states accessed concurrently by multiple threads in mutual exclusion according to the work pool paradigm.

6. Conclusions and future work

- 824 planning problems from real world applications and IPC.
- Experiments are carried out on a node with two processors with a total of 32 computing cores.
- Parallel algorithms solve 7% more problems (90%) than LPG-td (83%).
- PBFSD1 needed 224 s (sum of averages times), LPG-td required 463 s.

6. Conclusions and future work

- Parallel methods perform strongly in real-world application domains, solving up to 98% of the problems while LPG-td finds a solution in only 78% of them.
- **Parallel methods return higher quality solution (shorter plans at a fraction of the time taken by LPG-td).**
- PBFSD1 returns plans with 550 actions on average, PS (710 actions).

6. Conclusions and future work

- Mixed results in the IPC evaluation set: parallel methods perform strongly in 50% of the IPC set, where they return on average shorter plans more rapidly.
- LPG-td outperforms the proposed methods in the remaining 129 problems, solving 92% of problems; we notice that these instances are over-constrained;

➢ Solution space might not be large enough to justify parallelization.

➢ **Important observation since it might help to design pruning techniques on parallel branches to reduce computational overhead.**

6. Conclusions and future work

- AI Planning is a hard problem, where large search landscapes are neither categorized nor understood.
- Planning search spaces greatly vary among applications.
- Further research is needed on the potential application of parallel algorithms to domain-independent planning.
- In future work, we shall concentrate on the design and development of diversification and pruning techniques for parallel search exploration.

➢ We are developing alternative strategies for inserting and removing states from the global processing queues of our algorithms, including study of new data structure.

➢ We require greedy strategies that consider worse quality solutions to escape from local optima.

➢ Design efficient parallel methods for GPU computing accelerators.

➢ Application to new problems including real world problems in distributed modular robotic systems (see Li El Baz 2019).

References

- E. Burns, S. Lemons, W. Ruml, R. Zhou, Best-first heuristic search for multicore machines. Artif Intell Res 39, 2010, pp. 689–743.
- D. El Baz, B. Fakih, R. Sanchez Nigenda, V. Boyer. Parallel best-first search algorithms for planning problems on multi-core processors, The Journal of Supercomputing, Issue 3, February 2022, pp. 3122-3151.
- M. Evett, J. Hendler, A. Mahanti, D. Nau, Pra*: Massively parallel heuristic search. Journal of Parallel and Distributed Computing 25(2), 1995, pp. 133–143.
- A. Gerevini, A. Saetti, I. Serina, Planning through stochastic local search and temporal action graphs in LPG. Artif. Intell. Res., 20, 2003, pp. 239 – 290.

References

- Y Jinnai, A. Fukunaga, Abstract zobrist hashing: an efficient work distribution method for parallel best-first search. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI), 2016, pp 717–723.
- Y. Jinnai, A. Fukunaga, Automated creation of efficient work distribution functions for parallel best-first search. In: Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling (ICAPS), 2016, pp 184–192.
- Y. Jinnai, A. Fukunaga, On hash-based work distribution methods for parallel best-first search, J. Artif. Intell. Res. 60, 2017, pp. 491–548.
- A. Kishimoto, A. Fukunaga, A. Botea, Evaluation of a simple, scalable, parallel best-first search strategy. Artif Intell 195, 2013, pp. 222–248.

References

- R. Kuroiwa, A. Fukunaga, On the pathological search behavior of distributed greedy best-first search. In: Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling. 2019, pp 255–263.
- V. Vidal, A lookahead strategy for heuristic search planning, in: Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling, 2004, pp. 150–160.
- V. Vidal, L. Bordeaux, Y. Hamadi, Adaptive k-parallel best-first search: a simple but efficient algorithm for multi-core domain- independent planning. In: Proceedings of the Third Annual Symposium on Combinatorial Seach (SOC-10), 2010, pp 100–107
- L. Zhu, D. El Baz, A programmable actuator for combined motion and connection and its application to modular robot, Mechatronics, Vol. 58, April 2019, 9-19.

References

- M. Lalami, D. El Baz, GPU implementation of the Branch and bound method for knapsack problems, in Proceedings of the 26th IEEE Symposium IPDPSW 2012, Shanghai China, 20-25 May 2012, p. 1763-1771.
- D. El Baz, M. Elkihel, Load balancing methods and parallel dynamic programming algorithm using dominance technique applied to the 0-1 knapsack problem, Journal of Parallel and Distributed Computing, Vol. 65, 2005, p. 74-84.
- Adel Dabah, Ahcène Bendjoudi, Abdelhakim AitZai, Didier El Baz, Nadia Nouali Taboudjemat, Hybrid Multi-core CPU and GPU-based B&B Approaches for the Blocking Job Shop Scheduling Problem, Journal of Parallel and Distributed Computing, Juillet 2018, 117, 73-86.

References

- J. Luo, S. Fujimura, D. El Baz, B. Plazolles, GPU based parallel genetic algorithm for solving an energy efficient dynamic flexible flow shop scheduling problem, Journal of Parallel and Distributed Computing, Vol. 133, Novembre 2019, 244-257.
- J. Luo, D. El Baz, A dual heterogeneous island genetic algorithm for solving large size flexible flow shop scheduling problems on hybrid multi-core CPU and GPU platforms, Mathematical Problems in Engineering, 13 March 2019, 1-13.
- J. Luo, D. El Baz, R. Xue, J. Hu, Solving the dynamic energy aware job shop scheduling problem with the heterogeneous parallel genetic algorithm, Future Generation Computer Systems, Vol. 108, July 2020, p. 119–134.