Impact algoritme - Rijnland Optimalisatie

Impact tijdserie categorieën

De volgende Impact (Solution Path) time series worden door RTC-Tools gegenereerd, voor ieder gemaal:

- Water_quantity_impact
- Water_quality_impact
- Energy_cost_impact
- User_constraints_impact

GEM_Spaarndam_water_quantity_impact	PITimeSeries	
	locationId	GEM_Spaarndam
	parameterId	water_quantity_impact
GEM_Spaarndam_water_quality_impact	PITimeSeries	
	locationId	GEM_Spaarndam
	parameterId	water_quality_impact
GEM_Spaarndam_energy_cost_impact	PITimeSeries	
	locationId	GEM_Spaarndam
	parameterId	energy_cost_impact
GEM_Spaarndam_user_constraints_impact	PITimeSeries	
	locationId	GEM_Spaarndam
	parameterId	user_constraints_impact
GEM_Katwijk_water_quantity_impact	PITimeSeries	
	locationId	GEM_Katwijk
	parameterId	water_quantity_impact
GEM_Katwijk_water_quality_impact	PITimeSeries	
	locationId	GEM_Katwijk
	parameterId	water_quality_impact
GEM_Katwijk_energy_cost_impact	PITimeSeries	
	locationId	GEM_Katwijk
	parameterId	energy_cost_impact
GEM_Katwijk_user_constraints_impact	PITimeSeries	
	locationId	GEM_Katwijk
	parameterId	user_constraints_impact
GEM_Halfweg_water_quantity_impact	PITimeSeries	
	locationId	GEM_Halfweg
	parameterId	water_quantity_impact
GEM_Halfweg_water_quality_impact	PITimeSeries	
	locationId	GEM_Halfweg
	parameterId	water_quality_impact
GEM_Halfweg_energy_cost_impact	PITimeSeries	
	locationId	GEM_Halfweg
	parameterId	energy_cost_impact
GEM_Halfweg_user_constraints_impact	PITimeSeries	
	locationId	GEM_Halfweg
	parameterId	user_constraints_impact
GEM_Gouda_water_quantity_impact	PITimeSeries	
	locationId	GEM_Gouda
	parameterId	water_quantity_impact
GEM_Gouda_water_quality_impact	PITimeSeries	
	locationId	GEM_Gouda
	parameterId	water_quality_impact
GEM_Gouda_energy_cost_impact	PITimeSeries	
	locationId	GEM_Gouda
	parameterId	energy_cost_impact
GEM_Gouda_user_constraints_impact	PITimeSeries	
	locationId	GEM_Gouda
	parameterId	user_constraints_impact

We hebben in de huidige implementatie voor 3 gradaties gekozen (naast 0 – geen invloed). Mocht dit 2 gradaties moeten zijn, dan kan

*scaled_result = result * 3.0 / total_cost* veranderd worden naar ->

*scaled_result = result * 2.0 / total_cost*

Ter info: Voor het bepalen van de invloed van een bepaald doel op de gemaal inzet kunnen we verschillende doelen die tijdens dezelfde 'prioriteit' afgehandeld worden niet onderscheiden. Zo kunnen we de wind effecten impact (geïmplementeerd door middel van 'use_Hnode_range_goal') en water_quantity impact (RBP WaterLevelRangeGoal – 'use_H_range_goal'), binnenwaterstanden ('use_pump_Hdown_range_goal') dan ook niet onderscheiden, omdat ze beide tijdens de prioriteit 1 afgehandeld worden.

We hebben in plaats van een 'wind' categorie, een User constraints categorie toegevoegd – die de invloed van de door de gebruiker opgelegde afvoeren op de uiteindelijke oplossing vastgelegd. Al met al hebben we elke prioriteit aan een categorie 'toegeschreven' – zie de lijst in de algoritme beschrijving.

Classificering

Er is een waarde op de eerste tijdstap die de classificering weergeeft.

- Missing: geen informatie/doelen in die categorie niet actief
- 0: doel categorie had (vrijwel) geen invloed op de gemaal inzet
- 1: doel categorie had enige invloed op de gemaal inzet
- 2: doel categorie had redelijk wat invloed op de gemaal inzet
- 3: doel categorie domineerde de gemaal inzet

Beschrijving van strategie en Performance consequenties

Om de invloed van een prioriteit op de gemaal inzet te kunnen bepalen en dat vervolgens te kunnen vergelijken met de doorwerking bij opeenvolgende doelen hebben we aanvullende processing moeten introduceren. We hebben het algoritme zo 'modulair' mogelijk opgezet, zodat we dit in de toekomst wellicht uit kunnen splitsen naar een afzonderlijke file.

Simpel gezegd komt de strategie er op neer dat we de minimalisatie pompkosten/prioritering gemalen nu na elke afzonderlijke prioriteit uitvoeren (in plaats van aan het eind). Dit heeft als doel om het best mogelijke resultaat (geminimaliseerde pompkosten/prioritering) te krijgen – in de hypothetische situatie dat de optimalisatie gestopt zou zijn na die specifieke prioriteit.

Veranderingen in de gemaal inzet tussen deze geminimaliseerde 'tussenresultaten' geven dan een idee van de noodzaak van een bepaald doel om gemaal inzet aan te passen. Als een gemaal niet verandert, dan had dat doel blijkbaar geen invloed op het gemaal.

We kijken dan per gemaal naar de verhouding van de afvoer aanpassingen van prioriteit naar prioriteit (dus eigenlijk de verhouding tussen de verschillende categorieën). Deze verhouding bepaald dan welke categorie voor dat betreffende gemaal van de grootste invloed is.

Zonder deze afzonderlijke bepaling kun je de tussenresultaten niet vergelijken – dus het is echt noodzakelijk om dit te doen onder de gekozen opzet. Het gevolg is wel dat deze extra processing stappen een impact op de performance. Hoe groot die impact in absolute zin is, is moeilijk te voorspellen. We zijn aan het kijken of we deze extra stappen kunnen 'paralelliseren' – omdat ze

onafhankelijk van de bestaande optimalisatie probleem gedraaid kunnen worden. In het geval van meerdere cores (dit is eigenlijk altijd zo) zou dit dan geen rol meer mogen spelen. Dit is echter (nog) niet gerealiseerd. We willen dus graag feedback over de performance.

Algoritme

Hierbij de python code voor de impact tijdseries, inclusief gedetailleerde annotaties die beschrijven hoe de impact waardes tot stand komen (moet nog vertaald naar het Nederlands)

In de post() function

```
### Output Timeseries providing information on the impact of each pump at each prio ###

# Mapping of priorities to categories
# missing wind: its hard to isolate from quantity
pump_analysis_categories = ['water_quality', 'water_quantity', 'user_constraints', 'energy_cost']
priority_categories = {
    -5: 'water_quantity',
    0:  'water_quality',
    1:  'water_quantity',
    2:  'water_quality',
    10: 'water_quantity',
    20: 'water_quality',
    30: 'water_quantity',
    40: 'user_constraints'}

# We are only interested in the bozem pumps that can be optimized
optimized_pumps = self.one_way_pumps + self.two_way_pumps

# Initialize results dict
pump_analysis_results = {pump: {k: np.nan for k in pump_analysis_categories} for pump in optimized_pumps}

# Get the net cost of each pump. We simplify by taking the absolute
# value, but it should be accurate enough for the purpose of this
# analysis
pump_net_costs = {}
for pump in optimized_pumps:
    pump_net_costs[pump] = np.trapz(np.abs(self.get_timeseries(pump + '_minimized_cost').values), self.times())
total_cost = np.sum(pump_net_costs.values())

# Save pump cost contribution-
 these are directly proportional to final total cost
for pump in optimized_pumps:
    pump_analysis_results[pump]['energy_cost'] = pump_net_costs[pump]

# First check if pump analysis was enabled
if self.pumping_analysis in ['subprocess', 'serial']:

    # Wait until all subprocesses have stopped
    for p in self.pump_analysis_processes:
        p.join()

    # Determine proportional contribution of each pump
    proportion_contribution = {pump: pump_net_costs[pump] / total_cost for pump in optimized_pumps}

    # Dataframe contains the minimized pump cost at non-
pump priorities, with each pump separately listed
    df = pd.read_csv(os.path.join(self._output_folder, 'pump_costs.csv'))
```

```
46.
47.     # Make list of priorities. Pump priorities are excluded, and final
48.     # results of main optimization correspond to the last priority before
49.     # pump priorities. This is because pump priorities are minimization
50.     # goals applied to the decision variables, so they can't constrain
51.     # the objective but rather determine its final solution
52.     priorities = sorted([int(p) for p in df['priority']] + [self.previous_priority]
    )
53.
54.     # Configure dataframe
55.     df = df.set_index('priority')
56.     df = df[optimized_pumps]
57.
58.     # Set results of main optimization as the value at previous priority
59.     df.loc[self.previous_priority] = pump_net_costs
60.
61.     # Loop over priorities and set influence contributions. The final
62.     # contribution of a pump to a given analysis category is proportional
63.     # to the final results. However, we use the intermediate results to
64.     # determine the proportion of the final results that can be attributed
65.     # to each category. We we so this by adding up the cost that each goal
66.     # caused on each pump, but scaling by that pump's proportional
67.     # contribution to the final results. This is not perfect, but given
68.     # the limited data and runtime constraints it can give a useful
69.     # measure of the influence of each analysis category on each pump and
70.     # their relative impact on the final results
71.     for prev_prio, prio in zip(priorities[:-1], priorities[1:]):
72.         category = priority_categories[prio]
73.         delta_cost = np.sum(df.loc[prio]) - np.sum(df.loc[prev_prio])
74.         for pump in optimized_pumps:
75.             if np.isnan(pump_analysis_results[pump][category]):
76.                 pump_analysis_results[pump][category] = delta_cost * proportion_con
    tribution[pump]
77.             else:
78.                 pump_analysis_results[pump][category] += delta_cost * proportion_co
    ntribution[pump]
79.
80. # Set analysis results.
81. empty_array = np.full_like(self.times(),np.nan)
82. for pump, analysis_category in itertools.product(optimized_pumps, pump_analysis_cat
    egories):
83.     result = pump_analysis_results[pump][analysis_category]
84.     array = empty_array.copy()
85.     if not np.isnan(result):
86.         # If result is real, set the first element to the scaled result
87.         # We scale the impact number to be an integer from 0 to 3, rounding up for
    (0.25, 1.0]
88.         scaled_result = result * 3.0 / total_cost
89.         if (0.25 > scaled_result) and (scaled_result <= 1.0):
90.             scaled_result = 1.0
91.         else:
92.             scaled_result = round(scaled_result)
93.         array[0] = scaled_result
94.     self.set_timeseries('_'.join((pump, analysis_category, 'impact')), Timeseries(s
    elf.times(), array))
```

pumping_analysis.py

```
1.  import csv
2.  import os
3.  import numpy as np
4.  from casadi import constpow, sumRows, MX, MXFunction
5.  from rtctools.optimization.alias_tools import AliasDict
6.  from rtctools.optimization.caching import cached
7.  from rtctools.util import run_optimization_problem
```

```
8.  from rtctools.optimization.goal_programming_mixin import GoalProgrammingMixin
9.  import logging
10. logger = logging.getLogger("rtctools")
11.
12.
13. class PumpingAnalysis:
14.     def __init__(self, **kwargs):
15.         if self._first_run:
16.             self._results_are_current = False
17.         else:
18.             self._results_are_current = True
19.
20.         # Initialize results and seeding from main optimization
21.         _results = AliasDict(self.alias_relation)
22.         _results._d = self.class_results_d
23.         self._results = [_results]
24.
25.         class_seed = AliasDict(self.alias_relation)
26.         class_seed._d = self.class_seed_d
27.         self.class_seed = [class_seed]
28.
29.         self._subproblem_epsilons = []
30.         self._subproblem_path_epsilons = []
31.         self._subproblem_path_timeseries = []
32.         self._subproblem_objectives = []
33.
34.     def seed(self, ensemble_member=0):
35.         return self.class_seed[ensemble_member]
36.
37.     @cached
38.     def parameters(self, ensemble_member=0):
39.         parameters = AliasDict(self.alias_relation)
40.         parameters._d = self.class_parameters_d
41.         return parameters
42.
43.     def optimize(self, preprocessing=True, postprocessing=True, log_solver_failure_
    as_error=True):
44.         # Do pre-processing
45.         if preprocessing:
46.             self.pre()
47.
48.         # Initialize pump solution path file
49.         if self._first_run:
50.             with open(os.path.join(self._output_folder, 'pump_costs.csv'), 'wb') as
     csv_file:
51.                 writer = csv.writer(csv_file)
52.                 writer.writerow(['priority'] + sorted(self.pumps))
53.
54.         # Group goals into subproblems
55.         subproblems = []
56.         goals = self.goals()
57.         path_goals = self.path_goals()
58.
59.         pump_priorities = set(self.pump_priorities.values())
60.
61.         for priority in sorted(pump_priorities):
62.             subproblems.append((priority, [goal for goal in goals if int(goal.prior
    ity) == priority and not goal.is_empty], [
63.                                 goal for goal in path_goals if int(goal.priority) ==
     priority and not goal.is_empty]))
64.
65.         # Solve the subproblems one by one
66.         logger.info("PumpingAnalysis: Starting goal programming")
67.
68.         success = False
69.
```

```
70.         times = self.times()
71.
72.         options = self.goal_programming_options()
73.
74.
75.         for i, (priority, goals, path_goals) in enumerate(subproblems):
76.             logger.info("Solving goals at priority {}".format(priority))
77.
78.             # Reset epsilons
79.             self._subproblem_epsilons = []
80.             self._subproblem_path_epsilons = []
81.             self._subproblem_path_timeseries = []
82.
83.             # Reset objective function
84.             self._subproblem_objectives = []
85.             self._subproblem_path_objectives = []
86.
87.             for j, goal in enumerate(goals):
88.                 if goal.critical:
89.                     if not goal.has_target_bounds:
90.                         raise Exception("Minimization goals cannot be critical")
91.                     epsilon = 0.0
92.                 else:
93.                     if goal.has_target_bounds:
94.                         epsilon = MX.sym('eps_{}_{}'.format(i, j))
95.                         self._subproblem_epsilons.append(epsilon)
96.
97.                 if not goal.critical:
98.                     if goal.has_target_bounds:
99.                         self._subproblem_objectives.append(lambda problem, ensemble
    _member, goal=goal, epsilon=epsilon: goal.weight * constpow(
100.                                 problem.extra_variable(epsilon.getName(), ensemb
    le_member=ensemble_member), goal.order))
101.                         else:
102.                             self._subproblem_objectives.append(lambda problem, e
    nsemble_member, goal=goal: goal.weight * constpow(
103.                                 goal.function(problem, ensemble_member) / (goal.
    function_range[1] - goal.function_range[0]) / goal.function_nominal, goal.order))
104.
105.                     if goal.has_target_bounds:
106.                         for ensemble_member in range(self.ensemble_size):
107.                             self._add_goal_constraint(
108.                                 goal, epsilon, ensemble_member, options)
109.
110.                 for j, goal in enumerate(path_goals):
111.                     if goal.critical:
112.                         if not goal.has_target_bounds:
113.                             raise Exception("Minimization goals cannot be critic
    al")
114.                         epsilon = np.zeros(len(self.times()))
115.                     else:
116.                         if goal.has_target_bounds:
117.                             epsilon = MX.sym('path_eps_{}_{}'.format(i, j))
118.                             self._subproblem_path_epsilons.append(epsilon)
119.
120.                     if goal.has_target_min:
121.                         min_series = MX.sym('path_min_{}_{}'.format(i, j))
122.
123.                         if isinstance(goal.target_min, Timeseries):
124.                             target_min = Timeseries(goal.target_min.times, goal.
    target_min.values)
125.                             target_min.values[np.logical_or(np.isnan(target_min.
    values), np.isneginf(target_min.values))] = -sys.float_info.max
126.                         else:
127.                             target_min = goal.target_min
128.
```

```
                            self._subproblem_path_timeseries.append(
                                (min_series, target_min))
                        else:
                            min_series = None
                        if goal.has_target_max:
                            max_series = MX.sym('path_max_{}_{}'.format(i, j))

                            if isinstance(goal.target_max, Timeseries):
                                target_max = Timeseries(goal.target_max.times, goal.
    target_max.values)
                                target_max.values[np.logical_or(np.isnan(target_max.
    values), np.isposinf(target_max.values))] = sys.float_info.max
                            else:
                                target_max = goal.target_max

                            self._subproblem_path_timeseries.append(
                                (max_series, target_max))
                        else:
                            max_series = None

                        if not goal.critical:
                            if goal.has_target_bounds:
                                self._subproblem_objectives.append(lambda problem, e
    nsemble_member, goal=goal, epsilon=epsilon: goal.weight * sumRows(
                                    constpow(problem.state_vector(epsilon.getName(),
     ensemble_member=ensemble_member), goal.order)))
                            else:
                                self._subproblem_path_objectives.append(lambda probl
    em, ensemble_member, goal=goal: goal.weight *
                                    constpow(goal.function(problem, ensemble_member)
     / (goal.function_range[1] -
     goal.function_range[0]) / goal.function_nominal, goal.order))

                        if goal.has_target_bounds:
                            for ensemble_member in range(self.ensemble_size):
                                self._add_path_goal_constraint(
                                    goal, epsilon, ensemble_member, options, min_ser
    ies, max_series)

                    # Solve subproblem
                    success = super(GoalProgrammingMixin, self).optimize(
                        preprocessing=False, postprocessing=False, log_solver_failur
    e_as_error=log_solver_failure_as_error)
                    if not success:
                        break

                    self._first_run = False

                    # Store results.  Do this here, to make sure we have results eve
    n
                    # if a subsequent priority fails.
                    self._results_are_current = False
                    self._results = [self.extract_results(
                        ensemble_member) for ensemble_member in range(self.ensemble_
    size)]
                    self._results_are_current = True

                    # Re-add constraints, this time with epsilon values fixed
                    for ensemble_member in range(self.ensemble_size):
                        for j, goal in enumerate(goals):
                            if goal.critical:
                                continue

                            if not goal.has_target_bounds or goal.violation_timeseri
    es_id is not None or goal.function_value_timeseries_id is not None:
```

```
                                f = MXFunction('f', [self.solver_input], [goal.funct
    ion(self, ensemble_member)])
                                function_value = float(f([self.solver_output])[0])

                                # Store results
                                if goal.function_value_timeseries_id is not None:
                                    self.set_timeseries(goal.function_value_timeseri
    es_id, np.full_like(times, function_value), ensemble_member)

                            if goal.has_target_bounds:
                                epsilon = self._results[ensemble_member][
                                    'eps_{}_{}'.format(i, j)]

                                # Add a relaxation to appease the barrier method.
                                epsilon += options['constraint_relaxation']
                            else:
                                epsilon = function_value

                            # Add inequality constraint
                            self._add_goal_constraint(
                                goal, epsilon, ensemble_member, options)

                        for j, goal in enumerate(path_goals):
                            if goal.critical:
                                continue

                            if not goal.has_target_bounds or goal.violation_timeseri
    es_id is not None or goal.function_value_timeseries_id is not None:
                                # Compute path expression
                                expr = self.map_path_expression(goal.function(self,
    ensemble_member), ensemble_member)
                                f = MXFunction('f', [self.solver_input], [expr])
                                function_value = np.array(f([self.solver_output])[0]
    ).ravel()

                                # Store results
                                if goal.function_value_timeseries_id is not None:
                                    self.set_timeseries(goal.function_value_timeseri
    es_id, function_value, ensemble_member)

                            if goal.has_target_bounds:
                                epsilon = self._results[ensemble_member][
                                    'path_eps_{}_{}'.format(i, j)]

                                # Add a relaxation to appease the barrier method.
                                epsilon += options['constraint_relaxation']
                            else:
                                epsilon = function_value

                            # Add inequality constraint
                            self._add_path_goal_constraint(
                                goal, epsilon, ensemble_member, options)

                logger.info("Done goal programming")

                # Do post-processing
                if postprocessing:
                    self.post()

                # Done
                return success

            def post(self):
                # Determine if final run was successful
                successfull_return_strings = {'Solve_Succeeded', 'Solved_To_Acceptab
    le_Level'}
```

```
241.                success = self.solver_stats['return_status'] in successfull_return_s
    trings
242.                if not success:
243.                    logger.warning('What-
    if analysis falied at priority {}. Some solution path results will be missing.'.for
    mat(self.this_priority))
244.                else:
245.                    pump_costs = []
246.                    for pump in sorted(self.pumps):
247.                        try:
248.                            values = self.get_timeseries(pump + '_minimized_cost').v
    alues
249.                        except KeyError:
250.                            pump_costs.append('NaN')
251.                            continue
252.
253.                        pump_costs.append(np.trapz(np.abs(values), self.times()))
254.
255.                    with open(os.path.join(self._output_folder, 'pump_costs.csv'), '
    a+') as csv_file:
256.                        writer = csv.writer(csv_file)
257.                        writer.writerow([self.this_priority] + pump_costs)
258.
259.        # Get attibs from global scope (this file is run with execfile())
260.        attribs = globals().get('opt_checkpoint', None)
261.
262.        if attribs is None:
263.            logger.error('Missing data in pump analysis. Exiting.')
264.            print('Missing data in pump analysis. Exiting.')
265.            exit()
266.
267.        # Run pump cost analysis
268.        opt_problem = type('PumpingAnalysis', (PumpingAnalysis, Rijnland_Optimizatio
    n), attribs)
269.        run_optimization_problem(opt_problem, log_level=logging.ERROR)
```